The Dissertation Committee for Long Louis Ly
certifies that this is the approved version of the following dissertation:

Visibility Optimization for Autonomous Exploration and Surveillance-Evasion Games

Committee:

Yen-Hsi Tsai, Supervisor

Omar Ghattas

Ufuk Topcu

Paul Etienne Vouga

Rachel Ward

Visibility Optimization for Autonomous Exploration and Surveillance-Evasion Games

by

Long Louis Ly

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2020

To my family.

Acknowledgments

I am indebted to my advisor Richard Tsai for all his support and guidance throughout my graduate studies. Thanks to Joel Tropp for useful suggestions relating to the bounds in Chapter 2. This work was partially supported by the National Science Foundation (Grant No. DMS-1720171). Part of this research was performed while the author was visiting the Institute for Pure and Applied Mathematics (IPAM), which is supported by the National Science Foundation (Grant No. DMS-1440415). I acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing computational resources that have contributed to the research results reported within this document.

Visibility Optimization for Autonomous Exploration and Surveillance-Evasion Games

Long Louis Ly, Ph.D.
The University of Texas at Austin, 2020

Supervisor: Yen-Hsi Tsai

This dissertation considers problems involving line-of-sight visibility. In the exploration problem, the agent must efficiently map out a previously unknown environment, using as few sensor measurements as possible. For the surveillance-evasion game, the agent must always maintain visibility of a moving adversary. We first derive algorithms from a theoretical perspective. Although the resulting solutions are optimal, they are expensive to compute. We propose efficient approximations to the optimal solutions. At the expense of optimality, these approximations provide reasonable solutions that enable near real-time performance. We leverage state-of-the-art machine learning techniques to scale to scenarios that were not previously feasible. Level set functions allow for efficient computation of visibility and are a natural representation for input to convolutional neural networks.

Lastly, we consider the notion of the visibility of point clouds along rays. Using nearest neighbors along a set of randomly generated rays, we compute

a signature tensor which encodes geometric and statistical information about the point cloud. The signature, in combination with a convolutional neural network, enables efficient and robust classification of point clouds. We present promising results in 2D and 3D.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As we approach an era of autonomous drones and self-driving cars, it becomes increasingly important to develop efficient algorithms for path-planning and data processing. Depth sensors, such as LiDAR (Light Detection and Ranging), allow robotic agents to create virtual maps of the scene. Armed with this information, agents can perform complicated tasks in environments consisting of multiple obstacles. We consider two such problems involving line-of-sight visibility; two points are visible from one another if there is no obstacle in the line segment that connects the points. Lastly, we consider point cloud feature extraction and classification through the use of random rays.

1.1 Contributions

Broadly speaking, we develop mathematical algorithms for various problems involving visibility. Key components of these algorithms often involve computationally expensive operations. We apply neural networks to efficiently approximate those operations. Our success so far seems to be rooted in the generation of causally relevant data and in the flexibility of neural networks to interpolate such data.

In Chapter 2, we consider the exploration problem: an agent equipped with a depth sensor must map out a previously unknown environment using as few sensor measurements as possible. We propose an approach based on supervised learning of a greedy algorithm. We provide a bound on the optimality of the greedy algorithm using submodularity theory. Using a level set representation, we train a convolutional neural network to determine vantage points that maximize visibility. We show that this method drastically reduces the on-line computational cost and determines a small set of vantage points that solve the problem. This enables us to efficiently produce highly-resolved and topologically accurate maps of complex 3D environments. Unlike traditional next-best-view and frontier-based strategies, the proposed method accounts for geometric priors while evaluating potential vantage points. While existing deep learning approaches focus on obstacle avoidance and local navigation, our method aims at finding near-optimal solutions to the more global exploration problem. We present realistic simulations on 2D and 3D urban environments.

In Chapter 3, we consider surveillance-evasion differential games, where a pursuer must try to constantly maintain visibility of a moving evader. The pursuer loses as soon as the evader becomes occluded. Optimal controls for game can be formulated as a Hamilton-Jacobi-Isaac equation. We use an upwind scheme to compute the feedback value function, corresponding to the end-game time of the differential game. Although the value function enables optimal controls, it is prohibitively expensive to compute, even for a single pursuer and single evader on a small grid. We consider a discrete variant of

the surveillance-game. We propose two locally optimal strategies based on the static value function for the surveillance-evasion game with multiple pursuers and evaders. We show that Monte Carlo tree search and self-play reinforcement learning can train a deep neural network to generate reasonable strategies for on-line game play. Given enough computational resources and offline training time, the proposed model can continue to improve its policies and efficiently scale to higher resolutions.

In Chapter 4, we present an algorithm for the classification of point clouds. The algorithm exploits properties of the point clouds' signature in a new framework called *RaySense*. The RaySense signature is computed by finding nearest neighbors along a set of randomly generated rays. From the signature, statistical information about the whole data set, as well as certain geometric information, can be extracted, independent of the choice of ray set. A RaySense signature is not merely a subset of the point cloud: while all points sampled by each ray retain some local geometrical information of the point cloud, certain specific points are sampled repeatedly by different rays, giving a more global "sketch" of the point cloud's shape. We propose a convolutional neural network, *RayNN*, that uses RaySense signatures for point cloud classification. We evaluate RayNN on the 3D ModelNet benchmark and compare its performance with other state-of-the-art methods. RayNN results are comparable to other methods, while enjoying lower complexity. However, in the presence of additional corruptions, such as the introduction of unseen outliers or removal of data points, RayNN appears to be more robust.

1.2 Visibility level sets

We first review our representation of geometry and visibility. All the functions described below can be computed efficiently in $\mathcal{O}(m^d)$, where m is the number of grid points in each of d dimensions.

Level set functions

Level set functions [75, 85, 74] are useful as an implicit representation of geometry. Let $\Omega_{\text{obs}} \subseteq \mathbb{R}^d$ be a closed set of a finite number of connected components representing the obstacle. Denote the occluder function ϕ with the following properties:

$$\begin{cases} \phi(x) < 0 & x \in \Omega_{\text{obs}} \\ \phi(x) = 0 & x \in \partial\Omega_{\text{obs}} \\ \phi(x) > 0 & x \notin \Omega_{\text{obs}} \end{cases} \tag{1.1}$$

The occluder function is not unique; notice that for any constant $c > 0$, the function $c\phi$ also satisfies (1.1). We use the signed distance function as the occluder function:

$$\phi(x) := \begin{cases} -\inf\limits_{y \in \partial\Omega_{\text{obs}}} \|x - y\|_2 & x \in \Omega_{\text{obs}} \\ \inf\limits_{y \in \partial\Omega_{\text{obs}}} \|x - y\|_2 & x \notin \Omega_{\text{obs}} \end{cases} \tag{1.2}$$

The signed distance function is a viscosity solution to the Eikonal equation:

$$\begin{aligned} &|\nabla\phi| = 1 \\ &\phi(x) = 0 \text{ for } x \in \partial\Omega_{\text{obs}} \end{aligned} \tag{1.3}$$

It can be computed, for example, using the fast sweeping method [105] or the fast marching method [109].

Visibility function

Let Ω_{free} be the open set representing free space. Let $\mathcal{V}_{x_0}$ be the set of points in Ω_{free} visible from $x_0 \in \Omega_{\text{free}}$. We seek a function $\psi(x, x_0)$ with properties

$$\begin{cases} \psi(x,y) > 0 & x \in \mathcal{V}_{x_0} \\ \psi(x,y) = 0 & x \in \partial\mathcal{V}_{x_0} \\ \psi(x,y) < 0 & x \notin \mathcal{V}_{x_0} \end{cases} \tag{1.4}$$

Define the visibility level set function ψ:

$$\psi(x, x_0) = \min_{r \in [0,1]} \phi(x_0 + r(x - x_0)) \tag{1.5}$$

It can be computed efficiently using the fast sweeping method based on the PDE formulation described in [106]:

$$\begin{aligned} \nabla\psi \cdot \frac{x - x_0}{|x - x_0|} &= \min\left\{ H(\psi - \phi)\nabla\psi \cdot \frac{x - x_0}{|x - x_0|}, 0 \right\} \\ \psi(x_0, x_0) &= \phi(x_0) \end{aligned} \tag{1.6}$$

where H is the characteristic function of $[0, \infty)$.

Shadow function

When dealing with visibility, it is useful to represent the shadow regions. The gradient of the occluder function $\nabla\phi$ is perpendicular to the level sets ϕ. The dot product of $\nabla\phi$ and the viewing direction $(x_0 - x)$ characterizes the cosine of the grazing angle θ between obstacles and viewing direction. In particular, $|\theta| < \pi/4$ for the portion of obstacles that are directly visible to x_0.

Define the grazing function:

$$\gamma(x, x_0) = (x_0 - x)^T \cdot \nabla\phi(x) \tag{1.7}$$

By masking with the occluder function, we can characterize the portion of the obstacle boundary that is not visible from the vantage point x_0. Define the auxiliary and auxiliary visibility functions:

$$\begin{aligned} \alpha(x, x_0) &= \max\{\phi(x, x_0), \gamma(x, x_0)\} \\ \tilde{\alpha}(x, x_0) &= \min_{r\in[0,1]} \alpha(x + r(x_0 - x), x_0) \end{aligned} \tag{1.8}$$

By masking the auxilary visibility function with the obstacle, we arrive at the desired shadow function [101]:

$$\xi(x, x_0) = \max\{\tilde{\alpha}(x, x_0), -\phi(x)\} \tag{1.9}$$

The difference between the shadow function and the visibility function is that the shadow function excludes the obstacles. Although

$$\tilde{\xi}(x, x_0) = \max\{-\phi(x), \psi(x, x_0)\}$$

looks like a candidate shadow function, it is not correct. In particular

$$\{x|\tilde{\xi}(x, x_0) = 0\}$$

includes the portion of the obstacle boundary visible to x_0.

Figure 1.1 summarizes the relevant level set functions used in this work.

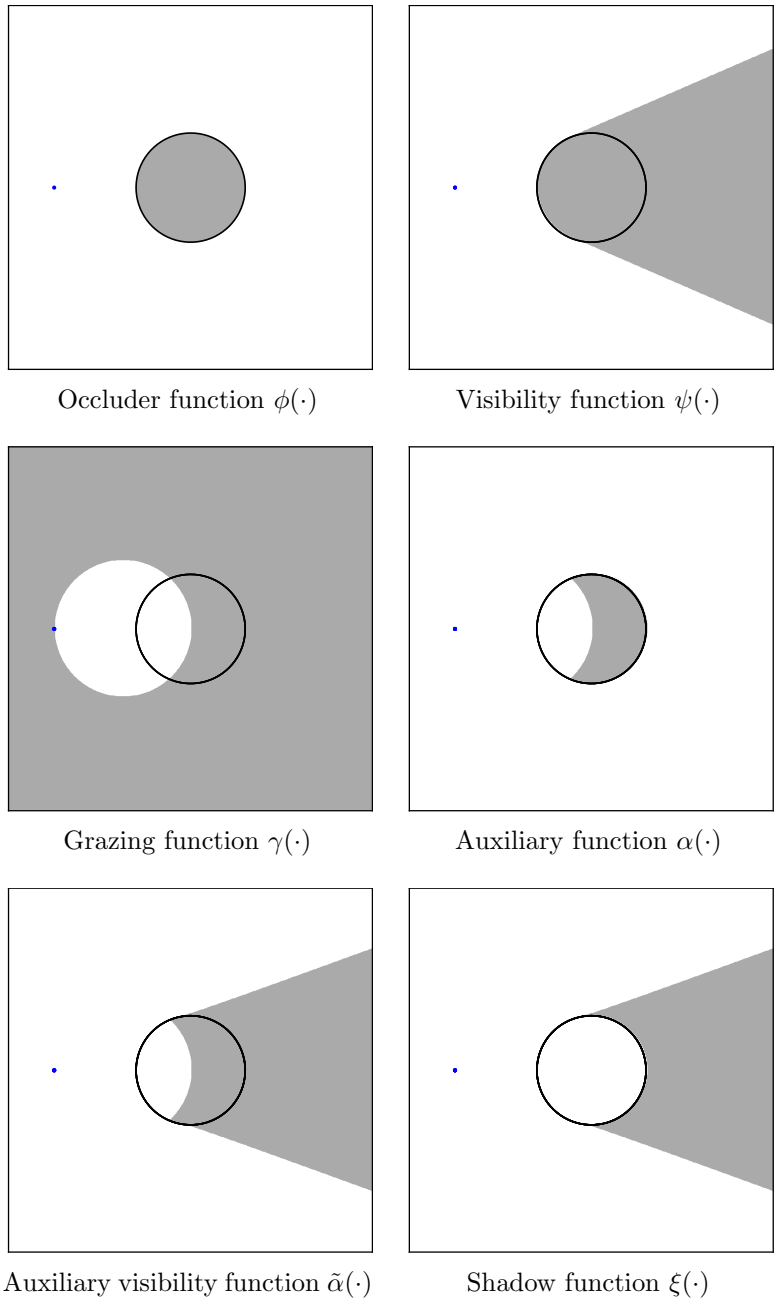

Figure 1.1: Level set functions from a vantage point x_0 (blue dot). Each function is negative in the shaded region. Obstacle boundary shown as black contour.

1.3 Convolutional neural networks

Convolutional neural networks (CNNs) have become the undisputed state-of-the-art for image classification tasks, where an input image has to be binned into one of k classes. CNNs are advantageous in that they use a learned classifier over learned set of features. This has been empirically shown to be better than the previous state-of-the-art, where learned classifiers were trained on hand-tuned features. Training generally requires lots of data and computational time.

A neural network architecture consists of various layers, which map the input to the desired output. There are many architectures; the most popular image classification networks are Alex-Net [52] , VGG [92], Inception [97], and ResNet [35]. More recently, fully convolutional neural networks have shown promise in dense inference tasks, which requires an output for each input. One such network, U-Net [79], is popular due to its simple architecture and ability to aggregate information across multiple scales.

Neural networks

We review some basic concepts. A single layer neural network is defined by a function

$$y = f(x; W, b) := \sigma(Wx + b),$$

where $x \in \mathbb{R}^m$ is the input and $y \in \mathbb{R}^n$ is the output. The parameters W and b are *learned*, where $W : \mathbb{R}^m \to \mathbb{R}^n$ is a linear function; i.e. an $n \times m$ real-valued matrix, while $b \in \mathbb{R}^n$ is a bias term. Lastly, $\sigma : \mathbb{R}^n \to \mathbb{R}^n$

is a nonlinear activation function. Common nonlinearities include *sigmoid*: $\sigma(y) = 1/(1 + e^{-y})$ and *ReLU*: $\sigma(y) = \max\{0, y\}$. The activation function is an "element-wise" function; i.e. if $y = (y_1, y_{2,} \cdots, y_n)$, then

$$\sigma(y) := (\sigma(y_1), \sigma(y_2), \cdots, \sigma(y_n)).$$

For classification tasks, the *softmax* function normalizes the output so that it sums to 1:

$$S(y) := \frac{e^y}{\sum_{j=1}^{n} e^{y_j}}.$$

Through the compositions of functions, a neural network can have multiple layers:

$$\begin{aligned} y^{(L)} &:= f_L(f_{L-1} \circ \cdots \circ f_1(x, W^{(1)}, b^{(1)}), \ldots, W^{(L)}, b^{(L)}) \\ &\equiv f(x; W^{(1)}, \cdots, W^{(L)}, b^{(1)}, \cdots, b^{(L)}), \end{aligned}$$

where

$$\begin{aligned} y^{(0)} &:= x, \\ y^{(\ell+1)} &:= \sigma_\ell(W^{(\ell+1)} y^{(\ell)} + b^{(\ell+1)}), \quad \ell = 0, 1, \cdots, L-1. \end{aligned} \tag{1.10}$$

The function σ_ℓ is the activation function or some operations that change the dimensionality of a vector for layer ℓ.

Residual networks

For very deep neural networks with many layers, [35] showed that incorporating a *residual connection* into (1.10) eases the training process and

improves accuracy. A *residual connection* with layer ℓ is similar to an explicit Euler scheme – the operation is an update to that layer rather than an operation on the entire output of that layer:

$$y^{(\ell+1)} := y^{(\ell)} + \sigma_\ell(W^{(\ell+1)}y^{(\ell)} + b^{(\ell+1)}), \;\; \ell = 0, 1, \cdots, L-1.$$

Training

The parameters $\theta := (W^{(1)}, \cdots, W^{(L))}, b^{(1)}, \cdots, b^{(L)})$, are learned via stochastic gradient descent to minimize the *empirical risk*:

$$\frac{1}{N}\sum_{j=1}^{N} \mathcal{L}(\tilde{y}_j, f(\tilde{x}_j; \theta))$$

where $\mathcal{L}$ is the loss function and $(\tilde{x}_j, \tilde{y}_j)\}_{j=1}^N$ are N training examples. The loss function is typically the *squared loss* for regression tasks:

$$\mathcal{L}(p, q) = \|p - q\|_2^2,$$

or the *cross entropy loss* for classification tasks:

$$\mathcal{L}(p, q) = -p \cdot \log(q).$$

Convolution

Next, we describe the operations that act on 2D image-based inputs. Generally, the input is a stack of 2D images. The third (stack) dimension is the feature dimension. One can think of each pixel in the image as a feature vector. For example, an RGB image has 3 channels, where each pixel is a vector describing the intensity of red, green, and blue.

A *convolution* layer convolves the input with a filter, whose weights are learned. The filter operates on all image channels, but only acts on a small neighborhood of spatial pixels, determined by the kernel size. Each filter takes as input a 3D image, and outputs a 2D image. Each convolutional layer may have multiple filters, resulting in a 3D stack of filtered images as output.

Let $x \in \mathbb{R}^{M_{\text{in}} \times N_{\text{in}} \times D_{\text{in}}}$ be an input vector-valued image, where M_{in}, N_{in}, and D_{in} are the width, height and number of channels, respectively. Let s_k be the kernel size (generally odd so that there is a well-defined center) and D_{out} be the number of filters. Then the output of the convolutional layer is $y \in \mathbb{R}^{M_{out} \times N_{out} \times D_{out}}$, where $M_{out} = M_{in} - s_k + 1$, $N_{out} = N_{in} - s_k + 1$, and each entry in $y := W * x + b$ is given by

$$y_{ijk} = b_k + \sum_{r=1}^{D_{in}} \sum_{q=0}^{s_k} \sum_{p=0}^{s_k} W_{p,q,r,k} \; x_{i+p,j+q,r}$$

where $W \in \mathbb{R}^{s_k \times s_k \times D_{in} \times D_{out}}$ and $b \in \mathbb{R}^{D_{out}}$ is the bias.

In general, a zero padding of s_p pixels can be introduced to the spatial dimensions of the input image to change the output size. This is particularly useful to make the output image the same size as the input. Also, a stride of s_s pixels can be used if it is not desirable to apply filter to consecutive pixels. In this case, $M_{out} = \frac{1}{s_s}(M_{in} - s_k + 2s_p) + 1$, $N_{out} = \frac{1}{s_s}(N_{in} - s_k + 2s_p) + 1$ and

$$y_{ijk} = b_k + \sum_{r=1}^{D_{in}} \sum_{q=0}^{s_k} \sum_{p=0}^{s_k} W_{p,q,r,k} \; \tilde{x}_{s_s \cdot i+p, s_s \cdot j+q, r}$$

where $\tilde{x}$ is the zero-padded image. A common choice for kernel size is $s_k = 3$, with zero padding $s_p = 1$ and stride $s_s = 1$.

Pooling

Pooling is a downsampling of the image, in order to reduce computational efforts and also allow the model to be spatially invariant. The most common is *max-pooling* with kernel size 2 and stride 2. This replaces each 2x2 patch in the input with its max value:

$$Y_{ijk} = \max_{p,q \in \{0,1\}^2} [X_{2 \cdot i+p, 2 \cdot j+q, k}]$$

For general kernel size s_k and stride s_s, we have

$$Y_{ijk} = \max_{p,q \in \{0,\ldots,s_k\}^2} [X_{s_s \cdot i+p, s_s \cdot j+q, k}]$$

Average pooling, where the max is replaced the average, is also common.

The convolution and pooling operations generalize to 3D images in a straight-forward manner; in those cases, the input will be a stack of 3D images.

Approximation theory

We review some literature relating to the capabilities of neural networks in approximating certain classes of functions. The term *hidden layer(s)* is used to refer to the intermediate layer(s) of a neural network. The number of *hidden units* refers to the dimension of the intermediate layers.

Universality

Define a sigmoidal function σ as any function with the properties: $\lim_{x \to -\infty} \sigma(x) = 0$ and $\lim_{x \to \infty} \sigma(x) = 1$. Cybenko [24] showed that neural

networks with one hidden layer and an arbitrary continuous sigmoidal function can approximate continuous functions $f : [0,1]^d \to \mathbb{R}$ to arbitrary precision. More precisely, let the neural network be a sum of the form

$$f_\theta(x) = \sum_{j=1}^{n} c_j \sigma(W_j^T x + b_j).$$

Then, given any $f \in C([0,1]^d)$ and $\varepsilon > 0$, there exists f_θ such that

$$|f_\theta(x) - f(x)| < \varepsilon \text{ for all } x \in [0,1]^d.$$

Similarly, Hornik et. al. [39] showed that a neural network, with as few as a single hidden layer, using an arbitrary nondecreasing sigmoidal function, is capable of approximating any Borel measurable function $f : \mathbb{R}^d \to \mathbb{R}^m$ to any level of accuracy. The work of [119] generalizes these ideas to convolutional neural networks.

Rate of convergence

However, the existence of a neural network that approximates a desired function well does not mean that there is a good way to constructing such a neural network, nor does it mean that the amount of computational resources needed for its construction is acceptable. The number of hidden units must be sufficiently large. A relevant question is how the number of hidden units n should be increased in order meet the accuracy ε and how it depends the input dimension d, the output dimension m, and the number of training samples N.

Barron [6, 7] gives such an analysis for a single hidden layer neural network approximating a class of functions $f : \mathbb{R}^d \to \mathbb{R}$ with bounded do-

mains. The analysis also depends on the first absolute moment of the Fourier transform of f:

$$C_{f,d} := \int_{\mathbb{R}^d} |\omega|_1 |\hat{f}(\omega)| d\omega.$$

They use a slightly different sigmoidal function σ which is Lipschitz and satisfies $\sigma(x) \to \pm 1$ at least polynomially fast as $x \to \pm\infty$, i.e., there exists $p > 0$ such that $\frac{\pm 1 - \sigma(x)}{|x|^p}$ is bounded as $x \to \pm\infty$. The result is as follows. Define the neural network

$$f_\theta(x) := \sum_{k=1}^{n} c_k \phi(W_k^T x + b_k) + c_0.$$

Then as $n, d, N \to \infty$, we have

$$||f - f_\theta||_2^2 \sim \mathcal{O}\Big(\frac{C_{f,d}^2}{n}\Big) + \mathcal{O}\Big(\frac{nd}{N} \log N\Big).$$

In particular, the second term on the right hand side increases as n increases. This says that increasing the network capacity (number of hidden units) leads to *overfitting* if the number of training examples does not increase accordingly. Meanwhile, the first term decreases as n increases. When

$$n \sim C_{f,d} \left(\frac{N}{d \log N}\right)^{1/2},$$

the optimal error is achieved:

$$||f - f_\theta||_2^2 \sim \mathcal{O}\Big(C_{f,d} \left(\frac{d}{N} \log N\right)^{1/2}\Big)$$

That is, the rate of convergence relative to the number of training examples is $\sqrt{\log N/N}$, with a rate $1/2$ that is independent of d. However, the constant $C_{f,d}$ may be exponential in d.

Chapter 2

Autonomous exploration

2.1 Introduction

We consider the problem of generating a minimal sequence of observing locations to achieve complete line-of-sight visibility coverage of an environment. In particular, we are interested in the case when environment is initially unknown. This is particularly useful for autonomous agents to map out unknown, or otherwise unreachable environments, such as undersea caverns. Military personnel may avoid dangerous situations by sending autonomous agents to scout new territory. We first assume the environment is known in order to gain insights.

Consider a domain $\Omega \subseteq \mathbb{R}^d$. Partition the domain $\Omega = \Omega_{\text{free}} \cup \Omega_{\text{obs}}$ into an open set Ω_{free} representing the free space, and a closed set Ω_{obs} of finite obstacles without holes. We will refer to the Ω_{obs} as the environment, since it is characterized by the obstacles. Let $x_i \in \Omega_{\text{free}}$ be a vantage point, from which a range sensor, such as LiDAR, takes omnidirectional measurements $\mathcal{P}_{x_i} : S^{d-1} \to \mathbb{R}$. That is, $\mathcal{P}_{x_i}$ outputs the distance to closest obstacle for each

This chapter, excluding sections 2.2.1 through 2.2.3, contains portions of work previously published in [66]. The problems and algorithms were developed jointly with my advisor and coauthor Richard Tsai. I was responsible for the numerical implementations.

direction in the unit sphere. One can map the range measurements to the visibility set $\mathcal{V}_{x_i}$; points in $\mathcal{V}_{x_i}$ are visible from x_i:

$$x \in \mathcal{V}_{x_i} \text{ if } \|x - x_i\|_2 < \mathcal{P}_{x_i}\Big(\frac{x - x_i}{\|x - x_i\|_2}\Big) \tag{2.1}$$

As more range measurements are acquired, Ω_{free} can be approximated by the *cumulatively visible set* Ω_k:

$$\Omega_k = \bigcup_{i=0}^{k} \mathcal{V}_{x_i} \tag{2.2}$$

By construction, Ω_k admits partial ordering: $\Omega_{i-1} \subset \Omega_i$. For suitable choices of x_i, it is possible that $\Omega_n \to \Omega_{\text{free}}$ (say, in the Hausdorff distance).

We aim at determining a *minimal set of vantage points* O from which every $x \in \Omega_{\text{free}}$ can be seen. One may formulate a constrained optimization problem and look for sparse solutions. When the environment is known, we have the *surveillance* problem:

$$\min_{O \subseteq \Omega_{\text{free}}} |O| \quad \text{subject to } \Omega_{\text{free}} = \bigcup_{x \in O} \mathcal{V}_x \ . \tag{2.3}$$

When the environment is not known apriori, the agent must be careful to avoid collision with obstacles. New vantage points must be a point that is currently visible. That is, $x_{k+1} \in \Omega_k$. Define the set of admissible sequences:

$$\mathbf{A}(\Omega_{\text{free}}) := \{(x_0, \ldots, x_{n-1}) \mid n \in \mathbb{N},\ x_0 \in \Omega_{\text{free}},\ x_{k+1} \in \Omega_k\}. \tag{2.4}$$

For the unknown environment, we have the *exploration* problem:

$$\min_{O \in \mathbf{A}(\Omega_{\text{free}})} |O| \quad \text{subject to } \Omega_{\text{free}} = \bigcup_{x \in O} \mathcal{V}_x. \tag{2.5}$$

The problem is feasible as long as obstacles do not have holes.

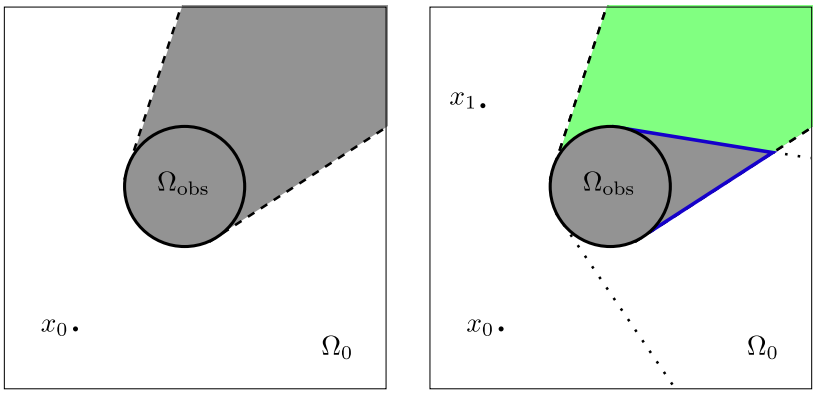

Figure 2.1: An illustration of the environment. Dashed and dotted lines are the horizons from x_0 and x_1, respectively. Their shadow boundary, B_1, is shown in thick, solid blue. The area of the green region represents $g(x_1; \Omega_0)$.

2.1.1 Related works

The surveillance problem is related to the art gallery problem in computational geometry, where the task is to determine the minimum set of guards who can together observe a polygonal gallery. Vertex guards must be stationed at the vertices of the polygon, while point guards can be anywhere in the interior. For simply-connected polygonal scenes, Chvátal showed that $\lfloor n/3 \rfloor$ vertex guards, where n is the number of vertices, are sometimes necessary and always sufficient [22]. For polygonal scenes with h holes, $\lfloor (n+h)/3 \rfloor$ point guards are sufficient [16, 38]. However, determining the optimal set of observers is NP-complete [110, 73, 61].

Goroshin et al. propose an alternating minimization scheme for optimizing the visibility of N observers [32]. Kang et al. use a system of differential

equations to optimize the location and orientation of N sensors to maximize surveillance [43]. Both works assume the number of sensors is given.

For the exploration problem, the "wall-following" strategy may be used to map out simple environments [117]. LaValle and Tovar et al. [102, 57, 104] combine wall-following with a gap navigation tree to keep track of gaps, critical events which hide a connected region of the environment that is occluded from a vantage point. Exploration is complete when all gaps have been eliminated. This approach does not produce any geometric representation of the environment upon completion, due to limited information from gap sensors.

A class of approaches pick new vantage points along shadow boundaries (aka frontiers), the boundary between free and occluded regions [116]. Ghosh et al. propose a frontier-based approach for 2D polygonal environments which requires $r+1$ views, where r is the number of reflex angles [29]. For general 2D environments, Landa et al. [56, 54, 55] use high order ENO interpolation to estimate curvature, which is then used to determine how far past the horizon to step. However, it is not necessarily optimal to pick only points along the shadow boundary, e.g. when the map is a star-shaped polygon [29].

Next-best-view algorithms try to find vantage points that maximize a utility function, consisting of some notion of *information gain* and another criteria such as path length. The vantage point does not have to lie along the shadow boundary. A common measure of information gain is the volume of *entire* unexplored region within sensor range that is not occluded by obstacles [31, 14, 15, 36]. Surmann et al. count the number of intersections of rays

into the occlusion [95], while Valente et al. [111] use the surface area of the shadow boundary, weighted by the viewing angle from the vantage points, to define potential information gain. The issue with these heurisitics is that they are independent of the underlying geometry. In addition, computing the information gain at each potential vantage point is costly and another heurisitic is used to determine which points to sample.

There has been some attempts to incorporate deep learning into the exploration problem, but they focus on navigation rather than exploration. The approach of Bai et al. [2] terminates when there is no occlusion within view of the agent, even if the global map is still incomplete. Tai and Liu [98, 99, 62] train agents to learn obstacle avoidance.

Our work uses a gain function to steer a greedy approach, similar to the next-best-view algorithms. However, our measure of information gain takes the geometry of the environment into account. By taking advantage of precomputation via convolutional neural networks, our model learns shape priors for a large class of obstacles and is efficient at runtime. We use a volumetric representation which can handle arbitrary geometries in 2D and 3D. Also, we assume that the sensor range is larger than the domain, which makes the problem more global and challenging.

2.2 Greedy algorithm

We propose a greedy approach which sequentially determines a new vantage point, x_{k+1}, based on the information gathered from all previous van-

tage points, $x_0, x_1, \cdots, x_k$. The strategy is greedy because x_{k+1} would be a location that *maximizes the information gain.*

For the surveillance problem, the environment is known. We define the *gain* function:

$$g(x; \Omega_k) := |\mathcal{V}_x \cup \Omega_k| - |\Omega_k|, \tag{2.6}$$

i.e. the volume of the region that is visible from x but not from $x_0, x_1, \cdots, x_k$. Note that g depends on Ω_{obs}, which we omit for clarity of notation. The next vantage point should be chosen to maximize the newly-surveyed volume. We define the greedy surveillance algorithm as:

$$x_{k+1} = \arg\max_{x \in \Omega_{\text{free}}} g(x; \Omega_k). \tag{2.7}$$

The problem of exploration is even more challenging since, by definition, the environment is not known. Subsequent vantage points must lie within the current visible set Ω_k. The corresponding greedy exploration algorithm is

$$x_{k+1} = \arg\max_{x \in \Omega_k} g(x; \Omega_k). \tag{2.8}$$

However, we remark that in practice, one is typically interested only in a subset $\mathcal{S}$ of all possible environments $\mathcal{S} := \{\Omega_{\text{obs}} | \Omega_{\text{obs}} \subseteq \mathbb{R}^d\}$.

For example, cities generally follow a grid-like pattern. Knowing these priors can help guide our estimate of g for certain types of Ω_{obs}, even when Ω_{obs} is unknown initially.

We propose to encode these priors formally into the parameters, θ, of a learned function:

$$g_\theta(x; \Omega_k, B_k) \text{ for } \Omega_{\text{obs}} \in \mathcal{S}, \tag{2.9}$$

where B_k is the part of $\partial\Omega_k$ that may actually lie in the free space Ω_{free}:

$$B_k = \partial\Omega_k \backslash \Omega_{\text{obs}}. \tag{2.10}$$

See Figure 2.2 for an example gain function. We shall demonstrate that while training for g_θ, incorporating the shadow boundaries helps, in some sense, localize the learning of g, and is essential in creating usable g_θ.

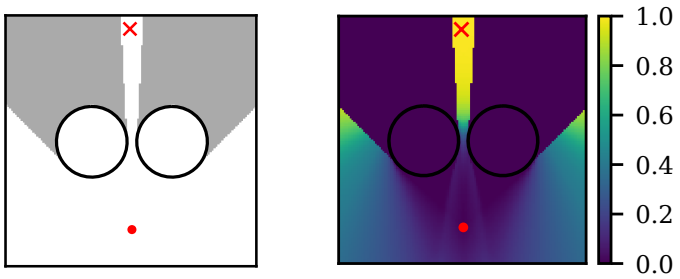

Figure 2.2: Left: the map of a scene consisting of two disks. Right: the intensity of the corresponding gain function. The current vantage point is shown as the red dot. The location which maximizes the gain function is shown as the red `x`.

2.2.1 A bound for the known environment

We present a bound on the optimality of the greedy algorithm, based on submodularity [50], a useful property of set functions. We start with standard definitions. Let V be a finite set and $f : 2^V \to \mathbb{R}$ be a set function which assigns a value to each subset $S \subseteq V$.

Definition 2.2.1. (Monotonicity) A set function f is *monotone* if for every $A \subseteq B \subseteq V$,

$$f(A) \leq f(B).$$

Definition 2.2.2. (Discrete derivative) The *discrete derivative* of f at S with respect to $v \in V$ is

$$\Delta_f(v|S) := f(S \cup \{v\}) - f(S).$$

Definition 2.2.3. (Submodularity) A set function f is *submodular* if for every $A \subseteq B \subseteq V$ and $v \in V \setminus B$,

$$\Delta_f(v|A) \geq \Delta_f(v|B).$$

In other words, set functions are submodular if they have diminishing returns. More details and extensions of submodularity can be found in [50].

Now, suppose the environment Ω_{obs} is known. Let O be the set of vantage points, and let $f(O)$ be the volume of the region visible from O:

$$\begin{aligned} \mathcal{V}(O) &:= \bigcup_{x \in O} \mathcal{V}_x \\ f(O) &:= \left| \mathcal{V}(O) \right| \end{aligned} \tag{2.11}$$

Lemma 2.2.1. *The function f is monotone.*

Proof. Consider $A \subseteq B \subseteq \Omega_{\text{free}}$. Since f is the cardinality of unions of sets,

we have

$$\begin{aligned} f(B) &= \Big| \bigcup_{x \in B} \mathcal{V}_x \Big| \\ &= \Big| \bigcup_{x \in A \cup \{B \setminus A\}} \mathcal{V}_x \Big| \\ &\geq \Big| \bigcup_{x \in A} \mathcal{V}_x \Big| \\ &= f(A). \end{aligned}$$

□

Lemma 2.2.2. *The function f is submodular.*

Proof. Suppose $A \subseteq B$ and $\{v\} \in \Omega_{\text{free}} \setminus B$. By properties of unions and intersections, we have

$$\begin{aligned} f(A \cup \{v\}) + f(B) &= \Big| \bigcup_{x \in (A \cup \{v\})} \mathcal{V}_x \Big| + \Big| \bigcup_{x \in B} \mathcal{V}_x \Big| \\ &\geq \Big| \bigcup_{x \in A \cup \{v\} \cup B} \mathcal{V}_x \Big| + \Big| \bigcup_{x \in (A \cup \{v\}) \cap B} \mathcal{V}_x \Big| \\ &= \Big| \bigcup_{x \in B \cup \{v\}} \mathcal{V}_x \Big| + \Big| \bigcup_{x \in A} \mathcal{V}_x \Big| \\ &= f(B \cup \{v\}) + f(A) \end{aligned}$$

Rearranging, we have

$$f(A \cup \{v\}) + f(B) \geq f(B \cup \{v\}) + f(A)$$

$$f(A \cup \{v\}) - f(A) \geq f(B \cup \{v\}) - f(B)$$

$$\Delta_f(v|A) \geq \Delta_f(v|B).$$

□

Submodularity and monotonicity enable a bound which compares the relative performance of the greedy algorithm to the optimal solution.

Theorem 2.2.3. *Let O_k^* be the optimal set of k sensors. Let $O_n = \{x_i\}_{i=1}^n$ be the set of n sensors placed using the greedy surveillance algorithm* (2.7)*. Then,*

$$f(O_n) \geq (1 - e^{-n/k}) f(O_k^*).$$

Proof. For $l < n$ we have

$$\begin{aligned}
f(O_k^*) &\leq f(O_k^* \cup O_l) && (2.12)\\
&= f(O_l) + \Delta_f(O_k^* | O_l) && (2.13)\\
&= f(O_l) + \sum_{i=1}^{k} \Delta_f(x_i^* | O_l \cup \{x_1^*, \ldots, x_{i-1}^*\}) && (2.14)\\
&\leq f(O_l) + \sum_{i=1}^{k} \Delta_f(x_i^* | O_l) && (2.15)\\
&\leq f(O_l) + \sum_{i=1}^{k} f(O_{l+1}) - f(O_l) && (2.16)\\
&= f(O_l) + k\big[f(O_{l+1}) - f(O_l)\big]. && (2.17)
\end{aligned}$$

Line (2.12) follows from monotonicity, (2.15) follows from submodularity of f, and (2.16) from definition of the greedy algorithm. Define $\delta_l := f(O_k^*) - f(O_l)$, with $\delta_0 := f(O_k^*)$. Then

$$\begin{aligned}
f(O_k^*) - f(O_l) &\leq k\big[f(O_{l+1}) - f(O_l)\big]\\
\delta_l &\leq k\big[\delta_l - \delta_{l+1}\big]\\
\delta_l\Big(1 - k\Big) &\leq -k\delta_{l+1}\\
\delta_l\Big(1 - \frac{1}{k}\Big) &\geq \delta_{l+1}
\end{aligned}$$

Expanding the recurrence relation with δ_n, we have

$$\begin{aligned}\delta_n &\leq \Big(1 - \frac{1}{k}\Big)\delta_{n-1} \\ &\leq \Big(1 - \frac{1}{k}\Big)^n \delta_0 \\ &= \Big(1 - \frac{1}{k}\Big)^n f(O_k^*)\end{aligned}$$

Finally, substituting back the definition for δ_n, we have the desired result:

$$\begin{aligned}\delta_n &\leq \Big(1 - \frac{1}{k}\Big)^n f(O_k^*) \\ f(O_k^*) - f(O_n) &\leq \Big(1 - \frac{1}{k}\Big)^n f(O_k^*) \\ f(O_k^*)\Big(1 - (1 - 1/k)^n\Big) &\leq f(O_n) \\ f(O_k^*)\Big(1 - e^{-n/k}\Big) &\leq f(O_n) \qquad (2.18)\end{aligned}$$

where (2.18) follows from the inequality $1 - x \leq e^{-x}$. □

In particular, if $n = k$, then $(1 - e^{-1}) \approx 0.63$. This means that k steps of the greedy algorithm is guaranteed to cover at least 63% of the total volume, if the optimal solution can also be obtained with k steps. When $n = 3k$, the greedy algorithm covers at least 95% of the total volume. In [71], it was shown that no polynomial time algorithm can achieve a better bound.

2.2.2 A bound for the unknown environment

When the environment is not known, subsequent vantage points must lie within the current visible set to avoid collision with obstacles:

$$x_{k+1} \in \mathcal{V}(O_k) \qquad (2.19)$$

Thus, the performance of the exploration algorithm has a strong dependence on the environment Ω_{obs} and the initial vantage point x_1. We characterize this dependence using the notion of the *exploration ratio*.

Given an environment Ω_{obs} and $A \subseteq \Omega_{\text{free}}$, consider the ratio of the marginal value of the greedy exploration algorithm, to that of the greedy surveillance algorithm:

$$\rho(A) := \frac{\sup_{x \in \mathcal{V}(A)} \Delta_f(x|A)}{\sup_{x \in \Omega_{\text{free}}} \Delta_f(x|A)}. \tag{2.20}$$

That is, $\rho(A)$ characterizes the relative gap (for lack of a better word) caused by the collision-avoidance constraint $x \in \mathcal{V}(A)$. Let $A_x = \{A \subseteq \Omega_{\text{free}} | x \in A\}$ be the set of vantage points which contain x. Define the *exploration ratio* as

$$\rho_x := \inf_{A \in A_x} \rho(A). \tag{2.21}$$

The exploration ratio is the worst-case gap between the two greedy algorithms, conditioned on x. It helps to provide a bound for the difference between the optimal solution set of size k, and the one prescribed by n steps the greedy exploration algorithm.

Theorem 2.2.4. *Let $O_k^* = \{x_i^*\}_{i=1}^k$ be the optimal sequence of k sensors which includes $x_1^* = x_1$. Let $O_n = \{x_i\}_{i=1}^n$ be the sequence of n sensors placed using the greedy exploration algorithm* (2.8)*. Then, for $k, n > 1$:*

$$f(O_n) \geq \Big[1 - \exp\Big(\frac{-(n-1)\rho_{x_1}}{k-1}\Big)\Big(1 - \frac{f(x_1)}{f(O_k^*)}\Big)\Big] f(O_k^*).$$

This is reminiscent of Theorem 2.2.3, with two subtle differences. The $\left[1-\frac{f(x_1)}{f(O_k^*)}\right]$ term accounts for the shared vantage point x_1. If $f(x_1)$ is large, then the exponential term has little effect, since $f(x_1)$ is already close to $f(O_k^*)$. On the other hand, if it is small, then the exploration ratio ρ_{x_1} plays a factor. The idea of the proof is similar, with some subtle differences in algebra to account for the shared vantage point x_1, and the exploration ratio ρ_{x_1}.

Proof. We have, for $l < n$:

$$
\begin{aligned}
f(O_k^*) &\leq f(O_k^* \cup O_l) \\
&= f(O_l) + \Delta_f(O_k^*|O_l) \\
&= f(O_l) + \sum_{i=1}^{k} \Delta_f(x_i^*|O_l \cup \{x_1^*, \ldots, x_{i-1}^*\}) && (2.22) \\
&\leq f(O_l) + \sum_{i=1}^{k} \Delta_f(x_i^*|O_l) && (2.23) \\
&= f(O_l) + \Delta_f(x_1^*|O_l) + \sum_{i=2}^{k} \Delta_f(x_i^*|O_l) \\
&= f(O_l) + \sum_{i=2}^{k} \Delta_f(x_i^*|O_l) && (2.24) \\
&\leq f(O_l) + \sum_{i=2}^{k} \max_{x \in \Omega_{\text{free}}} \Delta_f(x|O_l) \\
&\leq f(O_l) + \frac{1}{\rho_{x_1}} \sum_{i=2}^{k} \max_{x \in \mathcal{V}(O_l)} \Delta_f(x|O_l) && (2.25) \\
&\leq f(O_l) + \frac{1}{\rho_{x_1}} \sum_{i=2}^{k} f(O_{l+1}) - f(O_l) && (2.26) \\
&= f(O_l) + \frac{k-1}{\rho_{x_1}} \big[f(O_{l+1}) - f(O_l)\big].
\end{aligned}
$$

Line (2.22) is a telescoping sum, (2.23) follows from submodularity of f, (2.24) uses the fact that $x_1^* \in O_l$, (2.25) follows from the definition of ρ_{x_1} and (2.26) stems from the definition of the greedy exploration algorithm (2.8).

As before, define $\delta_l := f(O_k^*) - f(O_l)$. However, this time, note that $\delta_1 := f(O_k^*) - f(O_1) = f(O_k^*) - f(x_1)$. Then

$$\begin{aligned}
f(O_k^*) - f(O_l) &\leq \frac{k-1}{\rho_{x_1}}\big[f(O_{l+1}) - f(O_l)\big] \\
\delta_l &\leq \frac{k-1}{\rho_{x_1}}\big[\delta_l - \delta_{l+1}\big] \\
\delta_l\Big(1 - \frac{k-1}{\rho_{x_1}}\Big) &\leq -\frac{k-1}{\rho_{x_1}}\delta_{l+1} \\
\delta_l\Big(1 - \frac{\rho_{x_1}}{k-1}\Big) &\geq \delta_{l+1}
\end{aligned}$$

Expanding the recurrence relation with δ_n, we have

$$\begin{aligned}
\delta_n &\leq \Big(1 - \frac{\rho_{x_1}}{k-1}\Big)\delta_{n-1} \\
&\leq \Big(1 - \frac{\rho_{x_1}}{k-1}\Big)^{n-1}\delta_1 \\
&= \Big(1 - \frac{\rho_{x_1}}{k-1}\Big)^{n-1}\big[f(O_k^*) - f(x_1)\big]
\end{aligned}$$

Now, substituting back the definition for δ_n, we arrive at

$$\begin{aligned}
\delta_n &\leq \Big(1 - \frac{\rho_{x_1}}{k-1}\Big)^{n-1}\big[f(O_k^*) - f(x_1)\big] \\
f(O_k^*) - f(O_n) &\leq \Big(1 - \frac{\rho_{x_1}}{k-1}\Big)^{n-1}\big[f(O_k^*) - f(x_1)\big] \\
f(O_k^*) - f(x_1) - \big[f(O_n) - f(x_1)\big] &\leq \Big(1 - \frac{\rho_{x_1}}{k-1}\Big)^{n-1}\big[f(O_k^*) - f(x_1)\big] \\
\big[f(O_k^*) - f(x_1)\big]\Big(1 - \Big[1 - \frac{\rho_{x_1}}{k-1}\Big]^{n-1}\Big) &\leq \big[f(O_n) - f(x_1)\big] \\
\big[f(O_k^*) - f(x_1)\big]\Big(1 - e^{-\frac{(n-1)\rho_{x_1}}{k-1}}\Big) &\leq \big[f(O_n) - f(x_1)\big].
\end{aligned}$$

Finally, with some more algebra

$$
\begin{aligned}
\big[f(O_n) - f(x_1)\big] &\geq \Big(1 - e^{-\frac{(n-1)\rho_{x_1}}{k-1}}\Big)\big[f(O_k^*) - f(x_1)\big] \\
f(O_n) &\geq f(x_1) + \Big(1 - e^{-\frac{(n-1)\rho_{x_1}}{k-1}}\Big)\big[f(O_k^*) - f(x_1)\big] \\
f(O_n) &\geq f(x_1) + \Big(1 - e^{-\frac{(n-1)\rho_{x_1}}{k-1}}\Big)f(O_k^*) - f(x_1) + f(x_1)e^{-\frac{(n-1)\rho_{x_1}}{k-1}} \\
f(O_n) &\geq \Big(1 - e^{-\frac{(n-1)\rho_{x_1}}{k-1}}\Big)f(O_k^*) + f(x_1)e^{-\frac{(n-1)\rho_{x_1}}{k-1}} \\
f(O_n) &\geq \Big(1 - e^{-\frac{(n-1)\rho_{x_1}}{k-1}}\big[1 - \frac{f(x_1)}{f(O_n^*)}\big]\Big)f(O_k^*).
\end{aligned}
$$

□

Exploration ratio example

We demonstrate an example where ρ_x can be an arbitrarily small factor that is determined by the geometry of Ω_{free}. Figure 2.3 depicts an illustration of the setup for the narrow alley environment.

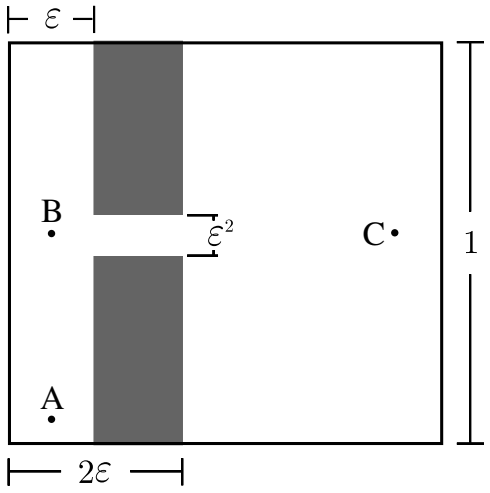

Figure 2.3: A map with a narrow alley. Scale exaggerated for illustration.

Consider a domain $\Omega = [0,1] \times [0,1]$ with a thin vertical wall of width $\varepsilon \ll 1$, whose center stretches from $(\frac{3}{2}\varepsilon, 0)$ to $(\frac{3}{2}\varepsilon, 1)$. A narrow opening of size $\varepsilon^2 \times \varepsilon$ is centered at $(\frac{3}{2}\varepsilon, \frac{1}{2})$. Suppose $x_1 = x_1^* = A$ so that

$$f(\{x_1\}) = \varepsilon + \mathcal{O}(\varepsilon^2),$$

where the ε^2 factor is due to the small sliver of the narrow alley visible from A. By observation, the optimal solution contains two vantage points. One such solution places $x_2^* = C$. The greedy exploration algorithm can only place $x_2 \in \mathcal{V}(x_1) = [0, \varepsilon] \times [0, 1]$. One possible location is $x_2 = B$. Then, after 2 steps of the greedy algorithm, we have

$$f(O_2) = \varepsilon + \mathcal{O}(\varepsilon^2).$$

Meanwhile, the total visible area is

$$f(O_2^*) = 1 - \mathcal{O}(\varepsilon)$$

and the ratio of greedy to optimal area coverage is

$$\frac{f(O_2)}{f(O_2^*)} = \frac{\varepsilon + \mathcal{O}(\varepsilon^2)}{1 - \mathcal{O}(\varepsilon)} = \mathcal{O}(\varepsilon) \tag{2.27}$$

The exploration ratio is $\rho_{x_1} = \mathcal{O}(\varepsilon^2)$, since

$$\begin{aligned} \max_{x \in \mathcal{V}(\{x_1\})} \Delta_f(x|\{x_1\}) &= \mathcal{O}(\varepsilon^2) \\ \max_{x \in \Omega_{\text{free}}} \Delta_f(x|\{x_1\}) &= 1 - \mathcal{O}(\varepsilon) \end{aligned} \tag{2.28}$$

According to the bound, with $k = n = 2$, we should have

$$\begin{aligned} \frac{f(O_2)}{f(O_2^*)} &\geq \left(1 - e^{-\frac{(n-1)\rho_{x_1}}{k-1}}\left[1 - \frac{f(x_1)}{f(O_2^*)}\right]\right) \\ &= \left(1 - e^{-\mathcal{O}(\varepsilon^2)}\left[1 - \mathcal{O}(\varepsilon)\right]\right) \\ &= \Omega(\varepsilon) \end{aligned} \tag{2.29}$$

which reflects what we see in (2.27).

On the other hand, if $O_2 = \{C, B\}$ and $O_2^* = \{C, B\}$, we would have

$$f(\{x_1\}) = 1 - \mathcal{O}(\varepsilon)$$

and $\rho_{x_1} = 1$, since both the greedy exploration and surveillance step coincide. According to the bound, with $k = n = 2$, we should have

$$\begin{aligned} \frac{f(O_2)}{f(O_2^*)} &\geq \left(1 - e^{-\frac{(n-1)\rho_{x_1}}{k-1}}\left[1 - \frac{f(x_1)}{f(O_n^*)}\right]\right) \\ &\geq 1 - \mathcal{O}(\varepsilon) \end{aligned} \tag{2.30}$$

which is the case, since $f(O_2) = f(O_2^*)$.

By considering the first vantage point x_1 as part of the bound, we account for some of the unavoidable uncertainties associated with unknown environments during exploration.

2.2.3 Numerical comparison

We compare both greedy algorithms on random arrangements of up to 6 circular obstacles. Each algorithm starts from the same initial position and runs until all free area is covered. We record the number of vantage points required over 200 runs for each number of obstacles.

Surprisingly, the exploration algorithm sometimes requires fewer vantage points than the surveillance algorithm. Perhaps the latter is too aggressive, or perhaps the collision-avoidance constraint acts as a regularizer. For example, when there is a single circle, the greedy surveillance algorithm places

the second vantage point x_2 on the opposite side of this obstacle. This may lead to two slivers of occlusion forming of either side of the circle, which will require 2 additional vantage points to cover. With the greedy exploration algorithm, we do not have this problem, due to the collision-avoidance constraint. Figure 2.4 shows an select example with 1 and 5 obstacles. Figure 2.5 show the histogram of the number of steps needed for each algorithm. On average, both algorithms require a similar number of steps, but the exploration algorithm has a slight advantage.

2.3 Learning the gain function

In this section, we discuss the method for approximating the gain function when the map is not known. Given the set of previously-visited vantage points, we compute the cumulative visibility and shadow boundaries. We approximate the gain function by applying the trained neural network on this pair of inputs, and pick the next point according to (2.7). This procedure repeats until there are no shadow boundaries or occlusions.

The data needed for the training and evaluation of g_θ are computed using level sets [75, 85, 74]. Occupancy grids may be applicable, but we choose level sets since they have proven to be accurate and robust. In particular, level sets are necessary for subpixel resolution of shadow boundaries and they allow for efficient visibility computation, which is crucial when generating the library of training examples.

The training geometry is embedded by a signed distance function, de-

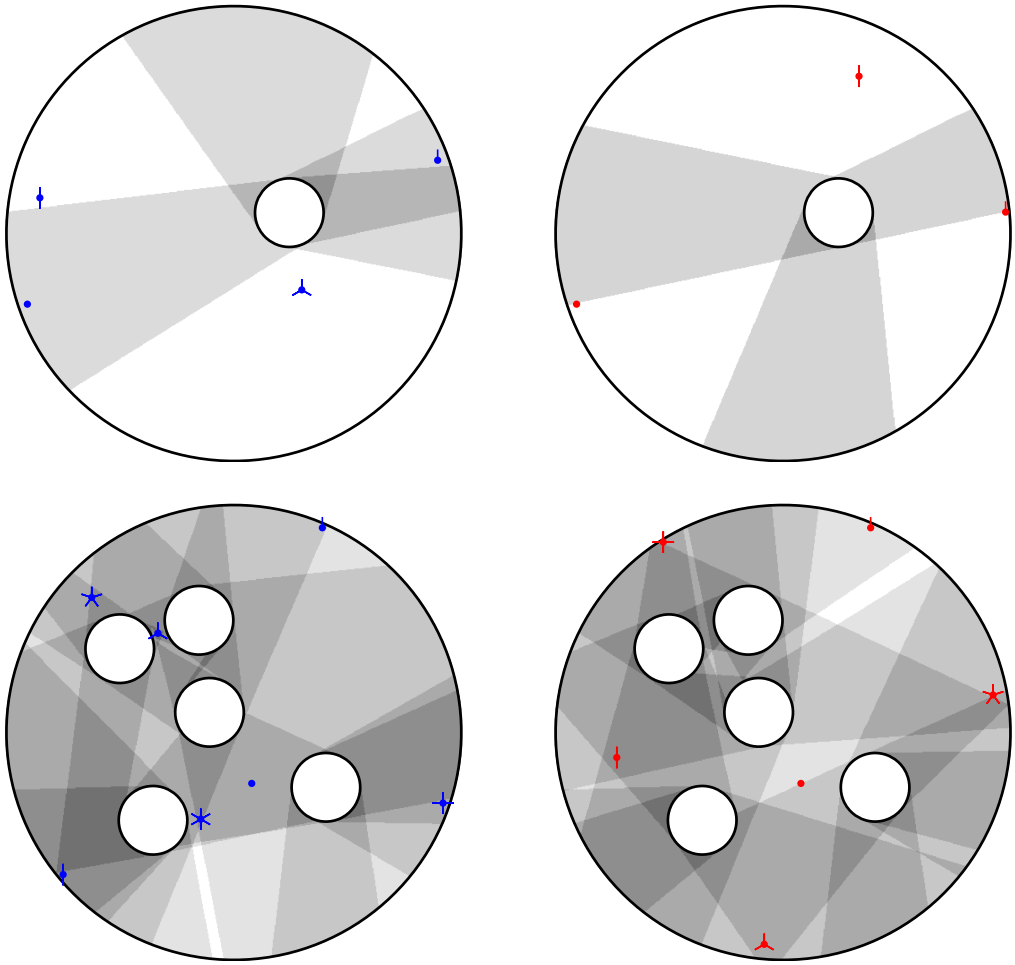

Figure 2.4: Comparing the greedy algorithm for the known (left) and unknown (right) environment on circular obstacles. Spikes on each vantage point indicate the ordering, e.g. the initial point has no spike. Gray areas are shadows from each vantage point. Lighter regions are visible from more vantage points.

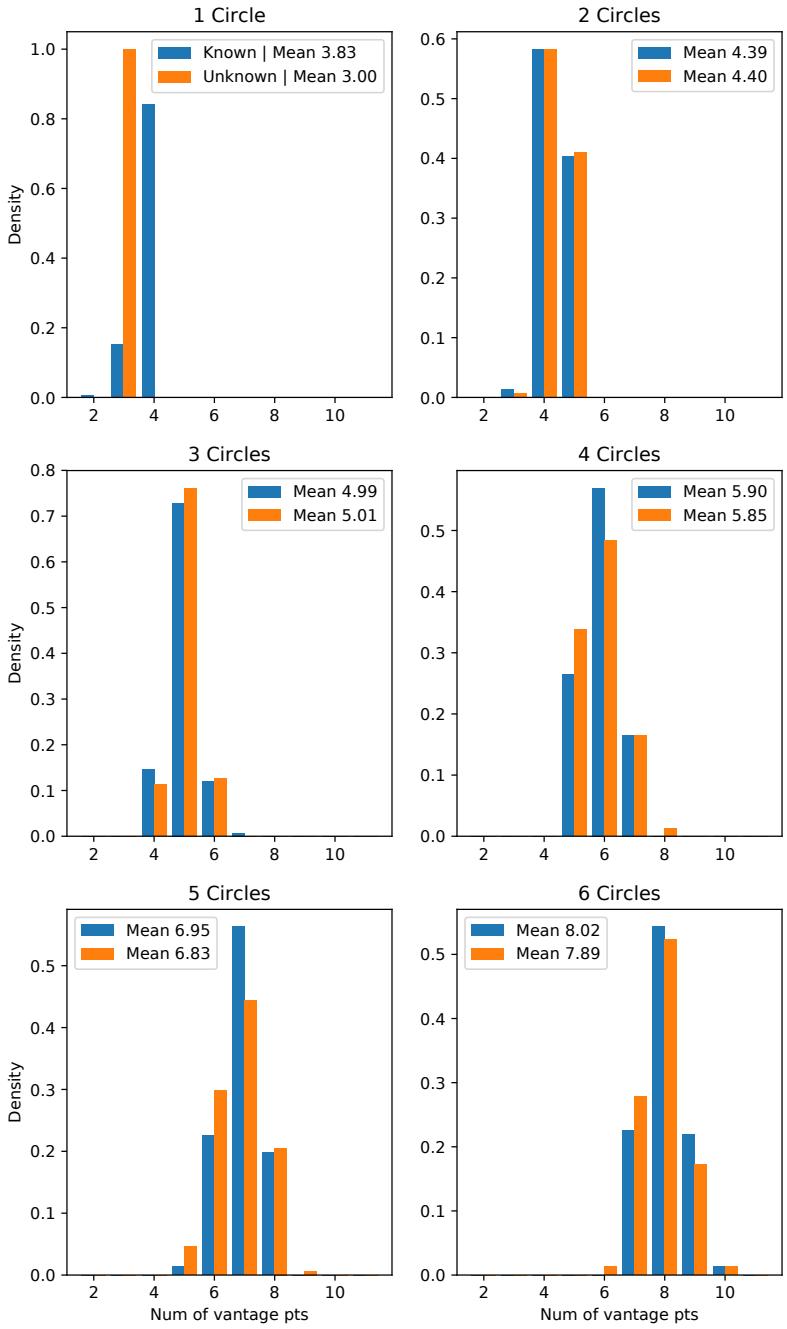

Figure 2.5: Histogram of number of vantage points needed for the surveillance (blue) and exploration (orange) greedy algorithms to completely cover environments consisting up of to 6 circles.

noted by ϕ. For each vantage point x_i, the visibility set is represented by the level set function $\psi(\cdot, x_i)$, which is computed efficiently using the algorithm described in [106].

In the calculus of level set functions, unions and intersections of sets are translated, respectively, into taking maximum and minimum of the corresponding characteristic functions. The cumulatively visible sets Ω_k are represented by the level set function $\Psi_k(x)$, which is defined recursively by

$$\Psi_0(x) = \psi(x, x_0), \tag{2.31}$$

$$\Psi_k(x) = \max\left\{\Psi_{k-1}(x), \psi(x, x_k)\right\}, \quad k = 1, 2, \ldots \tag{2.32}$$

where the max is taken point-wise. Thus we have

$$\Omega_{\text{free}} = \{x | \phi(x) > 0\}, \tag{2.33}$$

$$\mathcal{V}_{x_i} = \{x | \psi(x, x_i) > 0\}, \tag{2.34}$$

$$\Omega_k = \{x | \Psi_k(x) > 0\}. \tag{2.35}$$

The shadow boundaries B_k are approximated by the "smeared out" function:

$$b_k(x) := \delta_\varepsilon(\Psi_k) \cdot \left[1 - H(G_k(x))\right], \tag{2.36}$$

where $H(x)$ is the Heaviside function and

$$\delta_\varepsilon(x) = \frac{2}{\varepsilon} \cos^2\left(\frac{\pi x}{\varepsilon}\right) \cdot \mathbb{1}_{[-\frac{\varepsilon}{2}, \frac{\varepsilon}{2}]}(x), \tag{2.37}$$

$$\gamma(x, x_0) = (x_0 - x)^T \cdot \nabla\phi(x), \tag{2.38}$$

$$G_0 = \gamma(x, x_0), \tag{2.39}$$

$$G_k(x) = \max\{G_{k-1}(x), \gamma(x, x_k)\}, \quad k = 1, 2, \ldots \tag{2.40}$$

Recall, the shadow boundaries are the portion of the $\partial\Omega_k$ that lie in free space; the role of $1 - H(G_k)$ is to mask out the portion of obstacles that are currently visible from $\{x_i\}_{i=1}^k$. See Figure 1.1 for an example of γ. In our implementation, we take $\varepsilon = 3\Delta x$ where Δx is the grid node spacing. We refer the readers to [108] for a short review of relevant details.

When the environment Ω_{obs} is known, we can compute the gain function exactly

$$g(x; \Omega_k) = \int H\Big(H\big(\psi(\xi, x)\big) - H\big(\Psi_k(\xi)\big)\Big)\, d\xi. \tag{2.41}$$

We remark that the integrand will be 1 where the new vantage point uncovers something not previously seen. Computing g for all x is costly; each visibility and volume computation requires $\mathcal{O}(m^d)$ operations, and repeating this for all points in the domain results in $\mathcal{O}(m^{2d})$ total flops. We approximate it with a function $\tilde{g}_\theta$ parameterized by θ:

$$\tilde{g}_\theta(x; \Psi_k, \phi, b_k) \approx g(x; \Omega_k). \tag{2.42}$$

If the environment is unknown, we directly approximate the gain function by learning the parameters θ of a function

$$g_\theta(x; \Psi_k, b_k) \approx g(x; \Omega_k) H(\Psi_k) \tag{2.43}$$

using only the observations as input. Note the $H(\Psi_k)$ factor is needed for collision avoidance during exploration because it is not known *a priori* whether an occluded location y is part of an obstacle or free space. Thus $g_\theta(y)$ must be zero.

2.3.1 Training procedure

We sample the environments uniformly from a library. For each Ω_{obs}, a sequence of data pairs is generated and included into the training set $\mathcal{T}$:

$$\big(\{\Psi_k, b_k\}, g(x;\Omega_k)H(\Psi_k)\big), \qquad k = 0, 1, 2, \ldots. \tag{2.44}$$

For a given environment Ω_{obs}, define a path $O = \{x_i\}_{i=0}^{k}$ as admissible if $\phi(x_0) > 0$ and $\Psi_i(x_{i+1}) > 0$ for $i = 0, \ldots, k-1$. That is, it should only contain points in free space and in the case of exploration, subsequent points must be visible from at least one of the previous vantage points. Let $\mathcal{A}$ be the set of admissible paths. Then training set should ideally include all paths in $\mathcal{A}$. However this is too costly, since there are $\mathcal{O}(m^{kd})$ paths consisting of k steps. Instead, to generate causally relevant data, we use an ε-greedy approach: we uniformly sample initial positions. With probability ε, the next vantage point is chosen randomly from admissible set. With probability $1-\varepsilon$, the next vantage point is chosen according to (2.7). Figure 2.6 shows an illustration of the generation of causal data along the subspace of relevant shapes.

The function g_θ is learned by minimizing the empirical loss across all data pairs for each Ω_{obs} in the training set $\mathcal{T}$:

$$\underset{\theta}{\operatorname{argmin}} \frac{1}{N} \sum_{\Omega_{\text{obs}} \in \mathcal{T}} \sum_{k} L\Big(g_\theta(x;\Psi_k, b_k), g(x;\Omega_k)H(\Psi_k)\Big), \tag{2.45}$$

where N is the total number of data pairs. We use the cross entropy loss function:

$$L(p,q) = \int p(x)\log q(x) + (1-p(x))\log(1-q(x))\; dx. \tag{2.46}$$

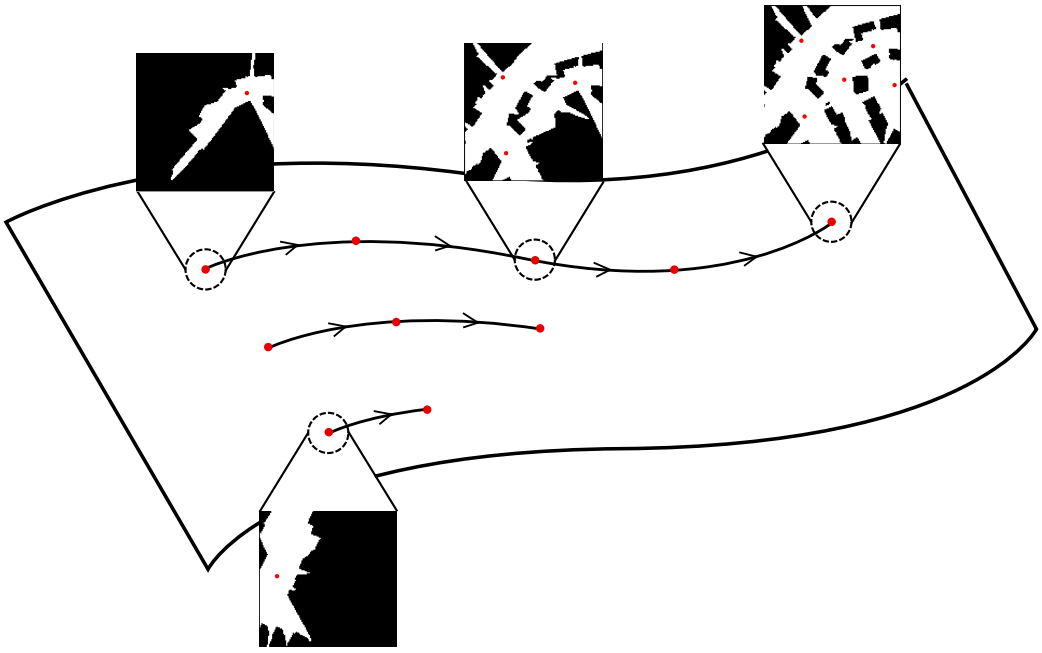

Figure 2.6: Causal data generation along the subspace of relevant shapes. Each dot is a data sample corresponding to a sequence of vantage points.

Network architecture

We use convolutional neural networks (CNNs) to approximate the gain function, which depends on the shape of Ω_{obs} and the location x. CNNs have been used to approximate functions of shapes effectively in many applications. Their feedforward evaluations are efficient if the off-line training cost is ignored. The gain function $g(x)$ does not depend *directly* on x, but rather, x's visibility of Ω_{free}, with a domain of dependence bounded by the sensor range. We employ a fully convolutional approach for learning g, which makes the network applicable to domains of different sizes. The generalization to 3D is also straight-forward.

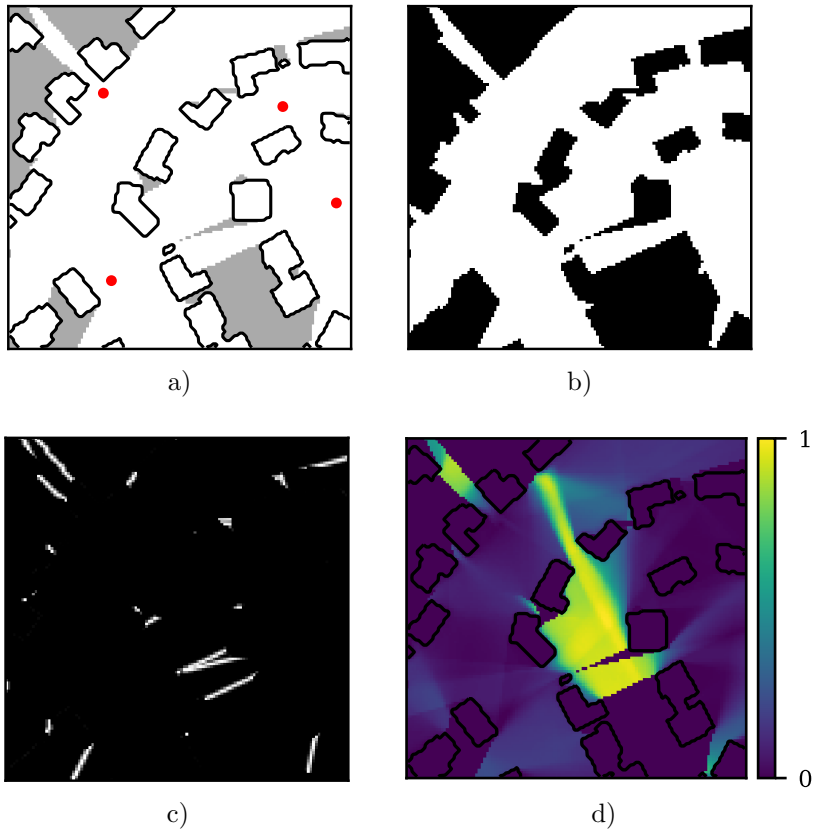

Figure 2.7: A training data pair consists of the cumulative visibility and shadow boundaries as input, and the gain function as the output. Each sequence of vantage points generates a data sample which depends strongly the shapes of the obstacles and shadows. a) The underlying map with current vantage points shown in red. b) The cumulative visibility of the current vantage points. c) The corresponding shadow boundaries. d) The corresponding gain function.

We base the architecture of the CNN on U-Net [79], which has had great success in dense inference problems, such as image segmentation. It aggregates information from various layers in order to have wide receptive fields while maintaining pixel precision. The main design choice is to make sure that the receptive field of our model is sufficient. That is, we want to make sure that the value predicted at each voxel depends on a sufficiently large neighborhood. For efficiency, we use convolution kernels of size 3 in each dimension. By stacking multiple layers, we can achieve large receptive fields. Thus the complexity for feedforward computations is linear in the total number of grid points.

Define a *conv block* as the following layers: convolution, batch norm, leaky `relu`, stride 2 convolution, batch norm, and leaky `relu`. Each *conv block* reduces the image size by a factor of 2. The latter half of the network increases the image size using *deconv blocks*: bilinear 2x upsampling, convolution, batch norm, and leaky `relu`.

Our 2D network uses 6 *conv blocks* followed by 6 *deconv blocks*, while our 3D network uses 5 of each block. We choose the number of blocks to ensure that the receptive field is at least the size of the training images: 128×128 and $64 \times 64 \times 64$. The first *conv block* outputs 4 channels. The number of channels doubles with each *conv block*, and halves with each *deconv block*.

The network ends with a single channel, kernel of size 1 convolution layer followed by the sigmoid activation. This ensures that the network aggregates all information into a prediction of the correct size and range.

2.4 Numerical results

We present some experiments to demonstrate the efficacy of our approach. Also, we demonstrate its limitations. First, we train on 128×128 aerial city blocks cropped from INRIA Aerial Image Labeling Dataset [68]. It contains binary images with building labels from several urban areas, including Austin, Chicago, Vienna, and Tyrol. We train on all the areas except Austin, which we hold out for evaluation. We call this model **City-CNN**. We train a similar model **NoSB-CNN** on the same training data, but omit the shadow boundary from the input. Third, we train another model **Radial-CNN**, on synthetically-generated radial maps, such as the one in Figure 2.13.

Given a map, we randomly select an initial location. In order to generate the sequence of vantage points, we apply (2.7), using g_θ in place of g. Ties are broken by choosing the closest point to x_k. We repeat this process until there are no shadow boundaries, the gain function is smaller than ϵ, or the residual is less than δ, where the residual is defined as:

$$r = \frac{|\Omega_{\text{free}} \setminus \Omega_k|}{|\Omega_{\text{free}}|}. \tag{2.47}$$

We compare these against the algorithm which uses the exact gain function, which we call **Exact**. We also compare against **Random**, a random walker, which chooses subsequent vantage points uniformly from the visible region, and **Random-SB** which samples points uniformly in a small neighborhood of the shadow boundaries. We analyze the number of steps required to cover the scene and the residual as a function of the number of steps.

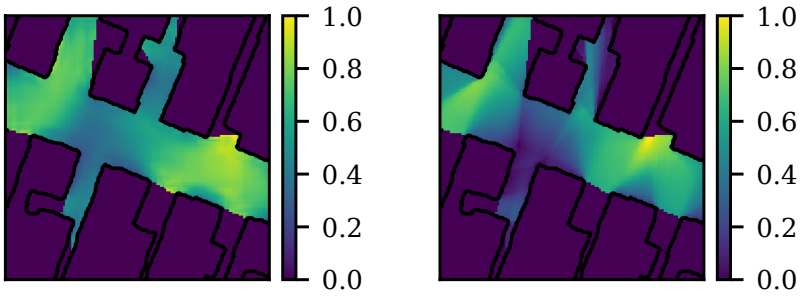

Figure 2.8: Comparison of predicted (left) and exact (right) gain function for an Austin map. Although the functions are not identical, the predicted gain function peaks in similar locations to the exact gain function, leading to similar steps.

Lastly, we present simulation for exploring 3D environments. Due to the limited availability of datasets, the model, **3D-CNN**, is trained using synthetic $64 \times 64 \times 64$ voxel images consisting of tetrahedrons, cylinders, ellipsoids, and cuboids of random positions, sizes, and orientations. In the site*, the interested reader may inspect the performance of the **3D-CNN** in some other challenging 3D environments.

For our experiments using trained networks, we make use of a CPU-only machine containing four Intel Core i5-7600 CPU @ 3.50GHz and 8 GB of RAM. Additionally, we use an Nvidia Tesla K40 GPU with 12 GB of memory for training and predicting the gain function in 3D scenes.

*http://visibility.page.link/demo

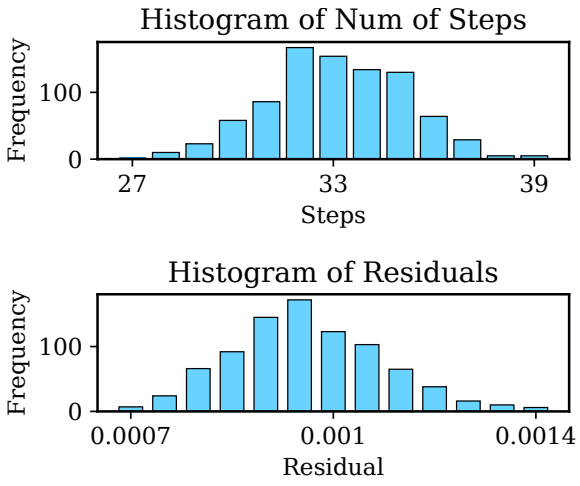

Figure 2.9: Distribution of the residual and number of steps generated across multiple runs over an Austin map. The proposed method is robust against varying initial conditions. The algorithm reduces the residual to roughly 0.1 % within 39 steps by using a threshold on the predicted gain function as a termination condition.

2D city

The **City-CNN** model works well on 2D Austin maps. First, we compare the predicted gain function to the exact gain function on a 128×128 map, as in Figure 2.8. Without knowing the underlying map, it is difficult to accurately determine the gain function. Still, the predicted gain function peaks in locations similar to those in the exact gain function. This results in similar sequences of vantage points.

The algorithm is robust to the initial positions. Figure 2.9 show the distribution of the number of steps and residual across over 800 runs from varying initial positions over a 512×512 Austin map. In practice, using the shadow

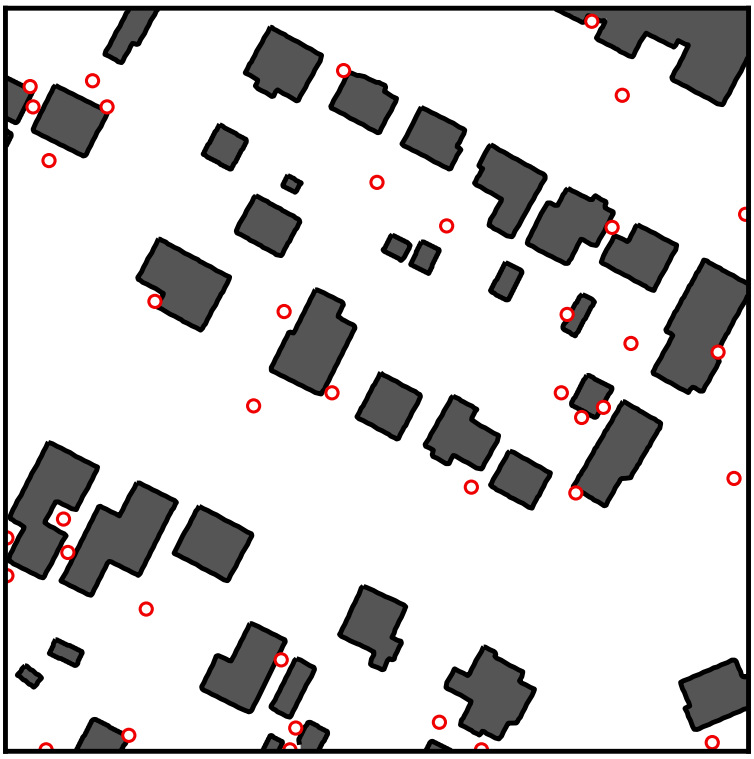

Figure 2.10: An example of 36 vantage points (red disks) using **City-CNN** model. White regions are free space while gray regions are occluded. Black borders indicate edges of obstacles.

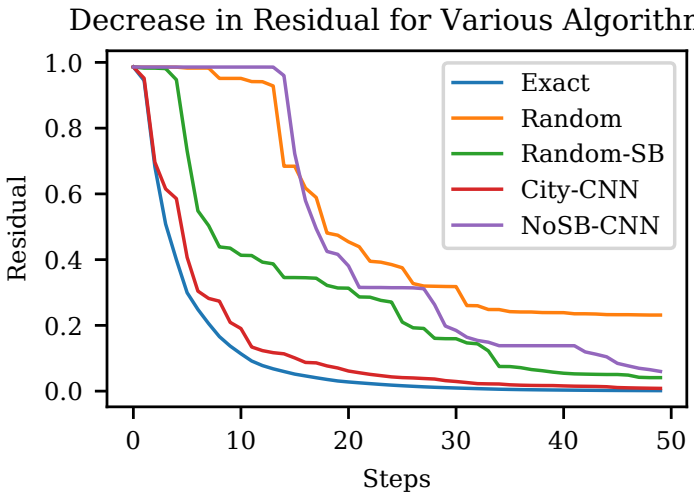

Figure 2.11: Graph showing the decrease in residual over 50 steps among various algorithms starting from the same initial position for an Austin map. Without using shadow boundary information, **NoSB-CNN** can at times be worse than **Random**. Our **City-CNN** model is significantly faster than **Exact** while remaining comparable in terms of residual.

boundaries as a stopping criteria can be unreliable. Due to numerical precision and discretization effects, the shadow boundaries may never completely disappear. Instead, the algorithm terminates when the maximum predicted gain falls below a certain threshold ϵ. In this example, we used $\epsilon = 0.1$. Empirically, this strategy is robust. On average, the algorithm required 33 vantage points to reduce the occluded region to within 0.1% of the explorable area.

Figure 2.10 shows an example sequence consisting of 36 vantage points. Each subsequent step is generated in under 1 sec using the CPU and instantaneously with a GPU.

Even when the maximizer of the predicted gain function is different

from that of the exact gain function, the difference in gain is negligible. This is evident when we see the residuals for **City-CNN** decrease at similar rates to **Exact**. Figure 2.11 demonstrates an example of the residual as a function of the number of steps for one such sequence generated by these algorithms on a 1024×1024 map of Austin. We see that **City-CNN** performs comparably to **Exact** approach in terms of residual. However, **City-CNN** takes 140 secs to generate 50 steps on the CPU while **Exact**, an $\mathcal{O}(m^4)$ algorithm, takes more than 16 hours to produce 50 steps.

Effect of shadow boundaries

The inclusion of the shadow boundaries as input to the CNN is critical for the algorithm to work. Without the shadow boundaries, the algorithm cannot distinguish between obstacles and occluded regions. If an edge corresponds to an occluded region, then choosing a nearby vantage point will reduce the residual. However, choosing a vantage point near a flat obstacle will result in no change to the cumulative visibility. At the next iteration, the input is same as the previous iteration, and the result will be the same; the algorithm becomes stuck in a cycle. To avoid this, we prevent vantage points from repeating by zeroing out the gain function at that point and recomputing the argmax. Still, the vantage points tend to cluster near flat edges, as in Figure 2.12. This clustering behavior causes the **NoSB-CNN** model to be, at times, worse than **Random**. See Figure 2.11 to see how the clustering inhibits the reduction in the residual.

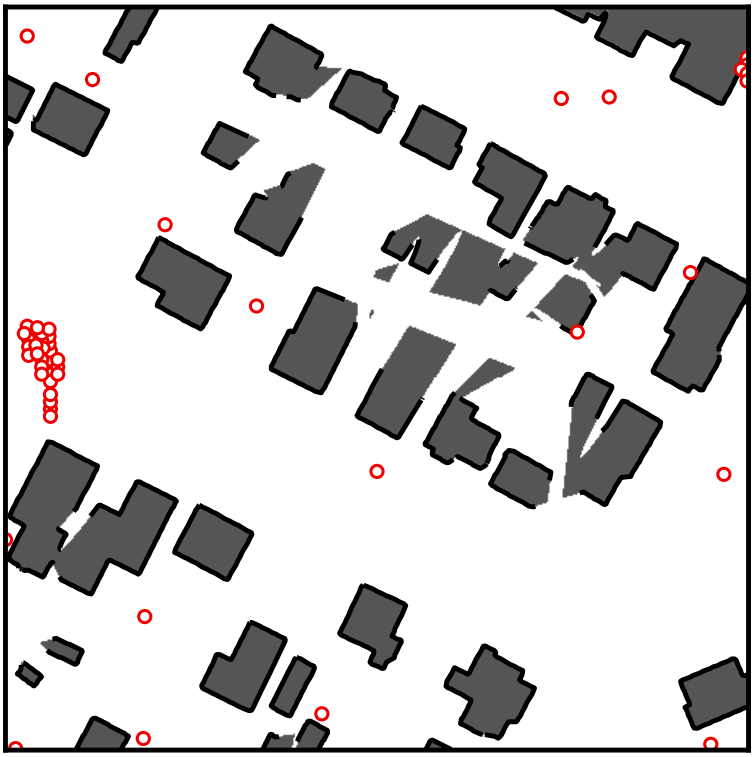

Figure 2.12: A sequence of 50 vantage points generated from **NoSB-CNN**. The points cluster near flat edges due to ambiguity and the algorithm becomes stuck. Gray regions without black borders have not been fully explored.

Effect of shape

The shape of the obstacles, i.e. Ω^c, used in training affects the gain function predictions. Figure 2.13 compares the gain functions produced by **City-CNN** and **Radial-CNN**.

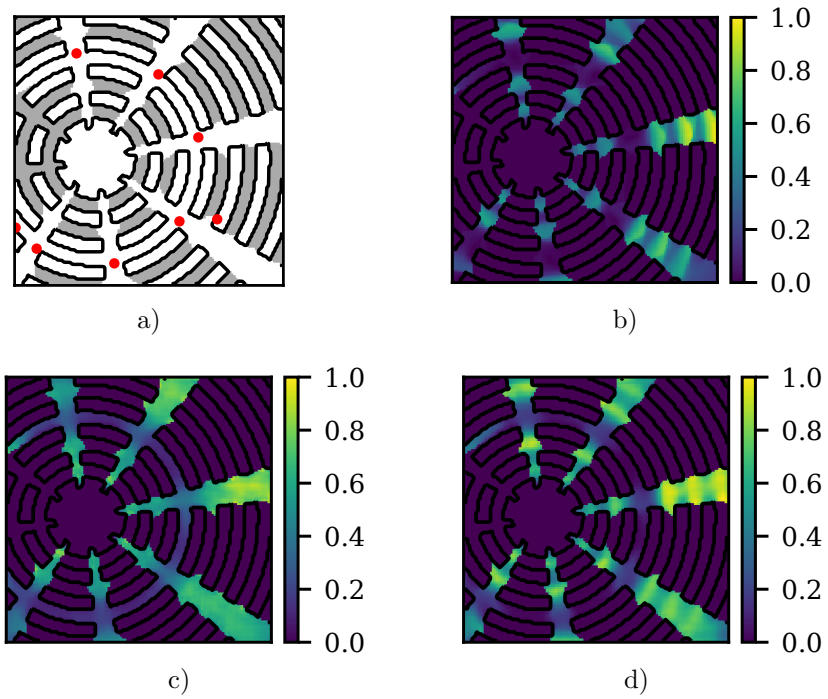

Figure 2.13: Comparison of gain functions produced with various models on a radial scene. Naturally, the CNN model trained on radial obstacles best approximates the true gain function. a) The underlying radial map with vantage points show in red. b) The exact gain function c) **City-CNN** predicted gain function. d) **Radial-CNN** predicted gain function.

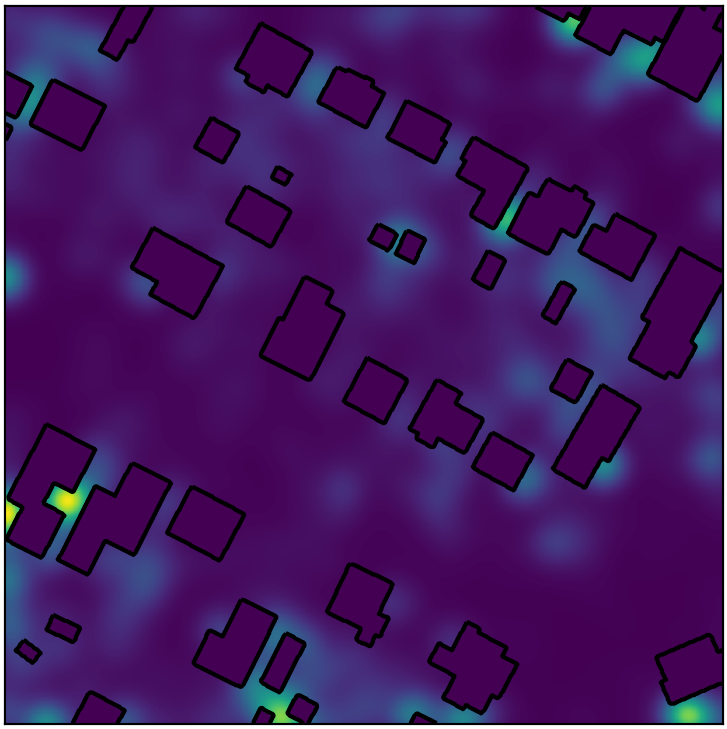

Figure 2.14: Distribution of vantage points generated by **City-CNN** method from various initial positions. Hot spots are brighter and are visited more frequently since they are essential for completing coverage.

Frequency map

Here we present one of our studies concerning the exclusivity of vantage point placements in Ω. We generated sequences of vantage points starting from over 800 different initial conditions using **City-CNN** model on a 512×512 Austin map. Then, we model each vantage point as a Gaussian with fixed width, and overlay the resulting distribution on the Austin map in Figure 2.14. This gives us a frequency map of the most recurring vantage points. These

hot spots reveal regions that are more secluded and therefore, the visibility of those regions is more sensitive to vantage point selection. The efficiency of the CNN method allows us to address many surveillance related questions for a large collection of relevant geometries.

Art gallery

Our proposed approach outperforms the computational geometry solution [72] to the art gallery problem, even though we do not assume the environment is known. The key issue with computational geometry approaches is that they are heavily dependent on the triangulation. In an extreme example, consider an art gallery that is a simple convex *n-gon*. Even though it is sufficient to place a single vantage point anywhere in the interior of the room, the triangulation-based approach produces a solution with $\lfloor n/3 \rfloor$ vertex guards.

Figure 2.15 shows an example gallery consisting of 58 vertices. The computational geometry approach requires $\lfloor \frac{n}{3} \rfloor = 19$ vantage points to completely cover the scene, even if point guards are used [16, 38]. The gallery contains $r = 19$ reflex angles, so the work of [29] requires $r + 1 = 20$ vantage points. On average, **City-CNN** requires only 8 vantage points.

3D environment

We present a 3D simulation of a 250m×250m environment based on Castle Square Parks in Boston. Figure 2.16 for snapshots of the algorithm in action. The map is discretized as a level set function on a $768 \times 768 \times 64$

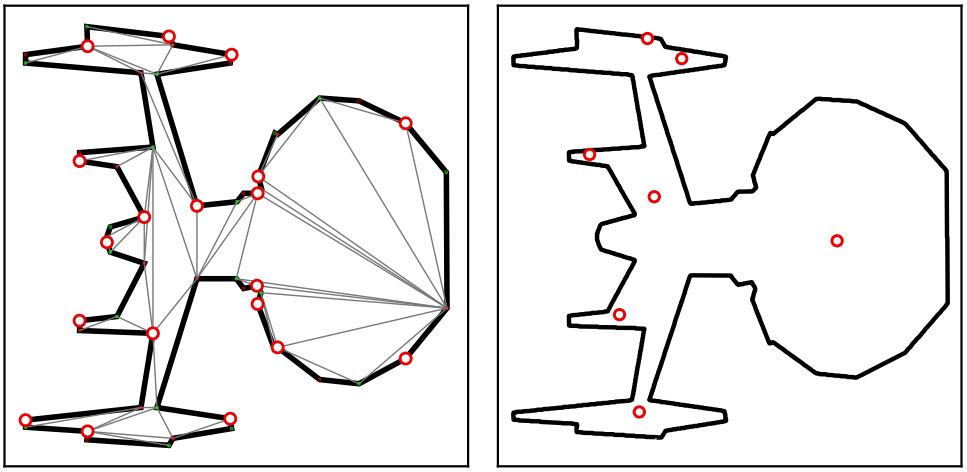

Figure 2.15: Comparison of the computational geometry approach and the **City-CNN** approach to the art gallery problem. The red circles are the vantage points computed by the methods. Left: A result computed by the computational geometry approach, given the environment. Right: An example sequence of 7 vantage points generated by the **City-CNN** model.

voxel grid. At this resolution, small pillars are accurately reconstructed by our exploration algorithm. Each step can be generated in 3 seconds using the GPU or 300 seconds using the CPU. Parallelization of the distance function computation will further reduce the computation time significantly. A map of this size was previously unfeasible. Lastly, Figure 2.17 shows snapshots from the exploration of a more challenging, cluttered 3D scene with many nooks.

2.5 Conclusion

From the perspective of inverse problems, we proposed a greedy algorithm for autonomous surveillance and exploration. We show that this formulation can be well-approximated using convolutional neural networks, which

learns geometric priors for a large class of obstacles. The inclusion of shadow boundaries, computed using the level set method, is crucial for the success of the algorithm. One of the advantages of using the gain function (2.6), an integral quantity, is its stability with respect to noise in positioning and sensor measurements. In practice, we envision that it can be used in conjuction with SLAM algorithms [25, 3] for a wide range of real-world applications.

One may also consider n-step greedy algorithms, where n vantage points are chosen simultaneously. However, being more greedy is not necessarily better. If the performance metric is the cardinality of the solution set, then it is not clear that multi-step greedy algorithms lead to smaller solutions. We saw in section 2.2 that, even for the single circular obstacle, the greedy surveillance algorithm may sometimes require more steps than the exploration algorithm to attain complete coverage.

If the performance metric is based on the rate in which the objective function increases, then a multi-step greedy approach would be appropriate. However, on a grid with m nodes in d dimensions, there are $\mathcal{O}(m^{nd})$ possible combinations. For each combination, computing the visibility and gain function requires $\mathcal{O}(nm^d)$ cost. In total, the complexity is $\mathcal{O}(nm^{d(n+1)})$, which is very expensive, even when used for offline training of a neural network. In such cases, it is necessary to selectively sample only the relevant combinations. One such way to do that, is through a tree search algorithm, which we discuss in the next section in the context of differential games.

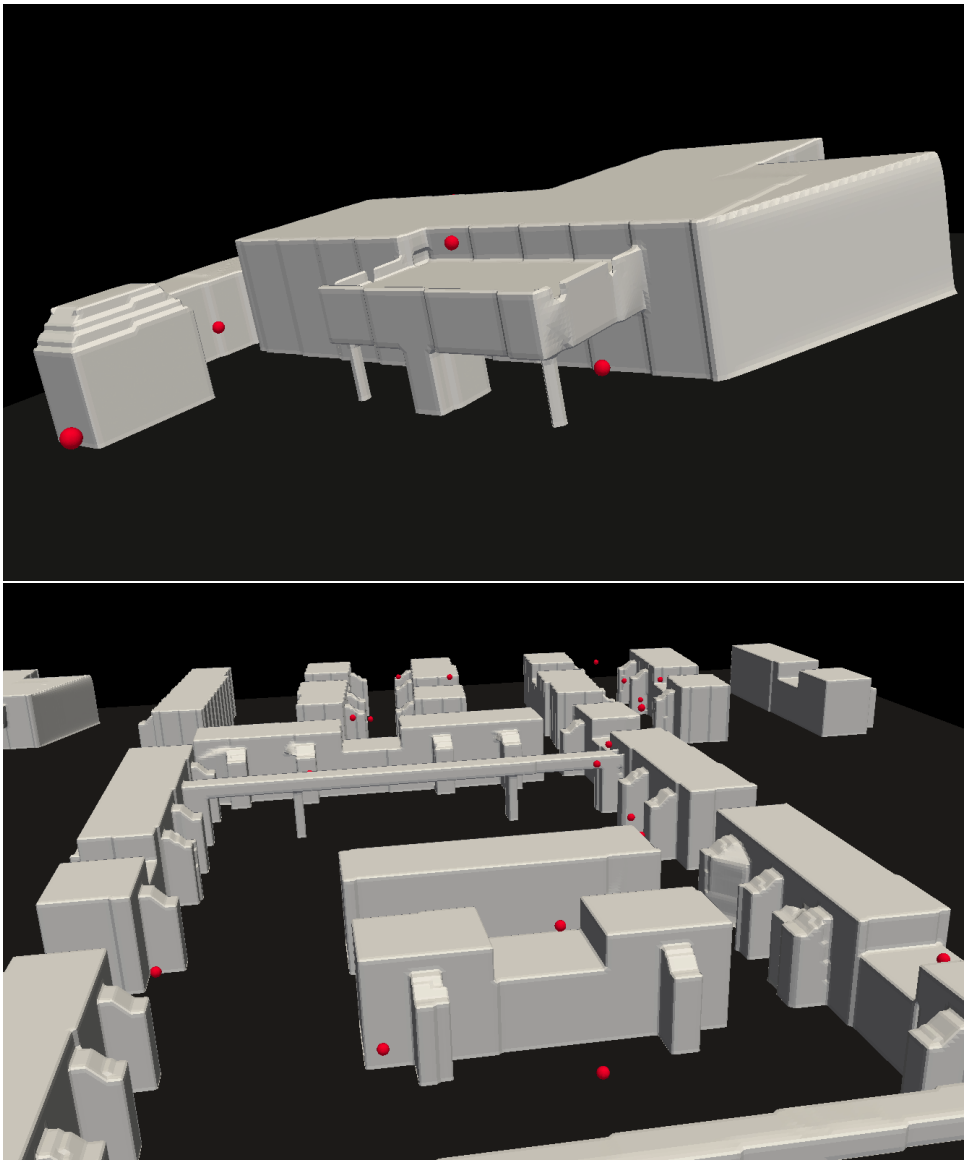

Figure 2.16: Snapshots demonstrating the exploration of an initially unknown 3D urban environment using sparse sensor measurements. The red spheres indicate the vantage point. The gray surface is the reconstruction of the environment based on line of sight measurements taken from the sequence of vantage points. New vantage points are computed in virtually real-time using **3D-CNN**.

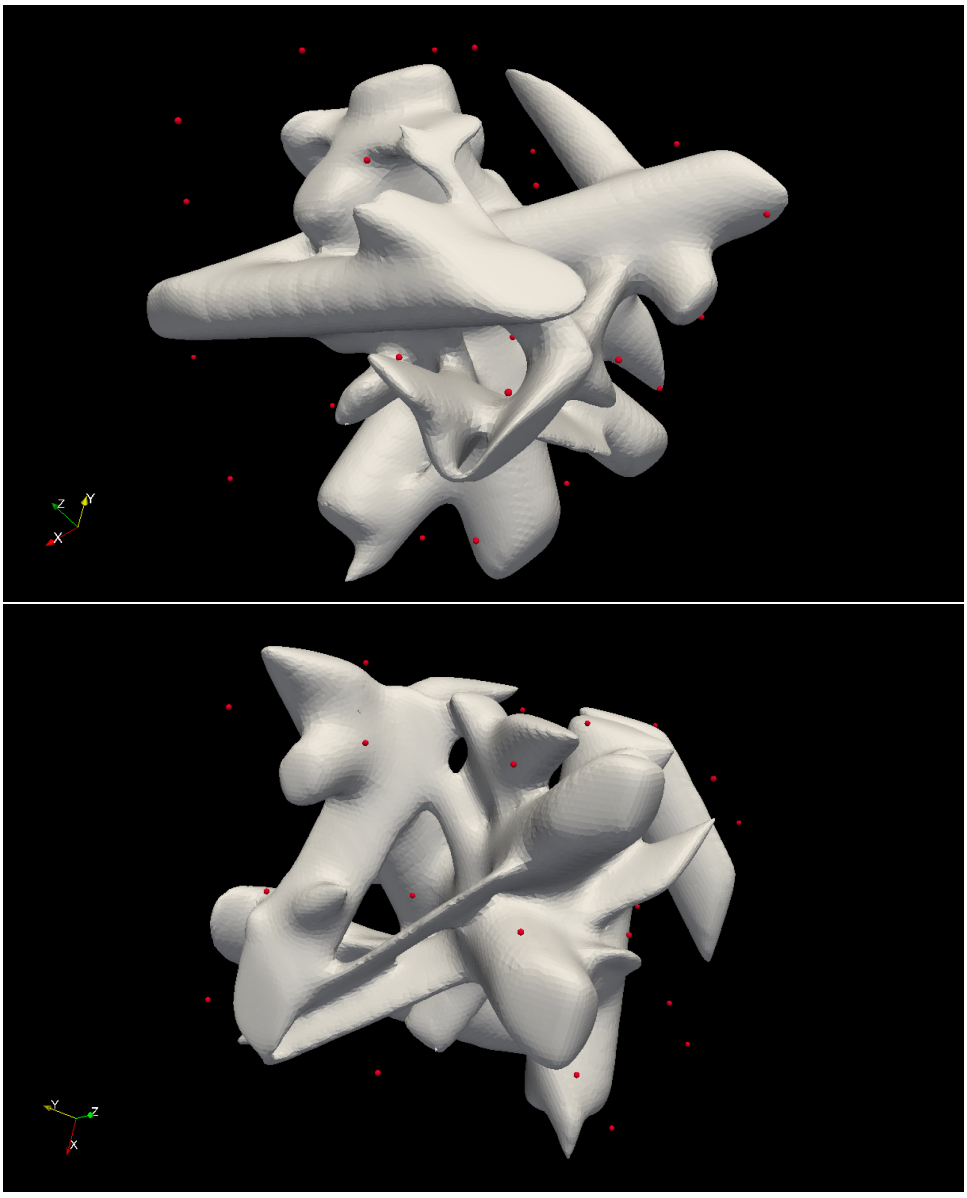

Figure 2.17: Snapshots of **3D-CNN** applied to exploration of a cluttered scene.

Chapter 3

Surveillance-evasion games

3.1 Introduction

We consider a multiplayer *surveillance-evasion* game consisting of two teams, the pursuers and the evaders. The pursuers must maintain line-of-sight visibility of the evaders for as long as possible as they move through an environment with obstacles. Meanwhile, the evaders aim to hide from the pursuers as soon as possible. The game ends when the pursuers lose sight of the evaders. We assume all players have perfect knowledge of the obstacles and the game is closed-loop – each player employs a feedback strategy, reacting dynamically to the positions of all other players.

In section 3.2, we consider the game in the context of Hamilton-Jacobi-Isaacs (HJI) equations. We propose a scheme to compute the value function, which, informally, describes how "good" it is for each player to be in a specific state. Then each player can pick the strategy that optimizes the value function locally. Due to the principal of optimality, local optimization with respect to the value function is globally optimal. This is because the value function encodes information from all possible trajectories. As a result, the value function is also very expensive to compute.

Section 3.3 discusses locally-optimal policies and section 3.4 presents search-based methods to learn policies for the multiplayer version of the game.

3.1.1 Related works

The surveillance-evasion game is related to a popular class of games called pursuit-evasion [37, 41], where the objective is for the pursuer to physically capture the evader. Classical problems take place in obstacle-free space with constraints on the players' motion. Variants include the lion and man [45, 87], where both players have the same maneuverability, and the homicidal chauffeur [69], where one player drives a vehicle, which is faster, but has constrained mobility. Lewin et. al. [63] considered a game in an obstacle-free space, where the pursuer must keep the evader within a detection circle.

Bardi et. al. [5] proposed a semi-Lagrangian scheme for approximating the value function of the pursuit-evasion game as viscosity solution to the Hamilton-Jacobi-Isaacs equation, in a bounded domain with no obstacles. In general, these methods are very expensive, with complexity $\mathcal{O}(m^{kd})$ where k is the number of players and d is the dimension. This is because the value function, once computed, can provide the optimal controls for all possible player positions. A class of methods try to deal with the curse of dimensionality by solving for the solutions of Hamilton-Jacobi equations at individual points in space and time. These methods are causality-free; the solution at one point does not depend on solutions at other points, making them conveniently parallelizable. They are efficient, since one only solves for the value function

locally, where it is needed, rather than globally. Chow et. al. [19, 20, 21] use the Hopf-Lax formula to efficiently solve Hamilton-Jacobi equations for a class of Hamiltonians. Sparse grid characteristics, due to Kang et. al. [44], is another causality-free method which finds the solution by solving a boundary value problem for each point. Unfortunately, these methods do not apply to domains with obstacles since they cannot handle boundary conditions.

The visibility-based pursuit-evasion game, introduced by Suzuki. et. al [96], is a version where the pursuer(s) must compute the shortest path to find all hidden evaders in a cluttered environment, or report it is not possible. The number of evaders is unknown and their speed is unbounded. Guibas et. al. [33] proposed a graph-based method for polygonal environments. Other settings include multiple pursuers [94], bounded speeds [103], unknown, piecewise-smooth planar environments [83], and simply-connected two-dimensional curved environments [59].

The surveillance-evasion game has been studied previously in the literature. LaValle et. al. [58] use dynamic programming to compute optimal trajectories for the pursuer, assuming a known evader trajectory. For the case of an unpredictable evader, they suggest a local heuristic: maximize the probably of visibility of the evader at the next time step. They also mention, but do not implement, an idea to locally maximize the evader's time to occlusion.

Bhattacharya et. al. [11, 13] used geometric arguments to partition the environment into several regions based on the outcome of the game. In [12, 122], they use geometry and optimal control to compute optimal trajecto-

ries for a single pursuer and single evader near the corners of a polygon. The controls are then extended to the whole domain containing polygonal obstacles by partitioning based on the corners [123], for the finite-horizon tracking problem [121], and for multiple players by allocating a pursuer for each evader via the Hungrarian matching algorithm [118].

Takei et. al. [101] proposed an efficient algorithm for computing the static value function corresponding to the open loop game, where each player moves according to a fixed strategy determined at initial time. Their open loop game is conservative towards the pursuer, since the evader can optimally counter any of the pursuer's strategies. As a consequence, the game is guaranteed to end in finite time, as long as the domain is not star-shaped. In contrast, a closed loop game allows players to react dynamically to each other's actions.

In [17, 30, 100], the authors propose optimal paths for an evader to reach a target destination, while minimizing exposure to an observer. In [30], the observer is stationary. In [17], the observer moves according to a fixed trajectory. In [100], the evader can tolerate brief moments of exposure so long as the consecutive exposure time does not exceed a given threshold. In all three cases, the observer's controls are restricted to choosing from a known distribution of trajectories; they are not allowed to move freely.

Bharadwaj et. al. [9] use reactive synthesis to determine the pursuer's controls for the surveillance-evasion game on a discrete grid. They propose a method of *belief abstraction* to coarsen the state space and only refine as needed. The method is quadratic in the number of states: $\mathcal{O}(m^{2kd})$ for k

players. While it is more computationally expensive than the Hamilton-Jacobi based methods, it is more flexible in being able to handle a wider class of temporal surveillance objectives, such as maintaining visibility at all times, maintaining a bound on the spatial uncertainty of the evader, or guaranteeing visibility of the evader infinitely often.

Recently, Silver et. al developed the AlphaGoZero and AlphaZero programs that excel at playing Go, Chess, and Shogi, without using any prior knowledge of the games besides the rules [90, 89]. They use Monte Carlo tree search, deep neural networks and self-play reinforcement learning to become competitive with the world's top professional players.

Contributions

We use a Godunov upwind scheme to compute the value function for the closed loop surveillance-evasion game with obstacles in two dimensions. The state space is four dimensional. The value function allows us to compute the optimal feedback controls for the pursuers and evaders. Unlike the static game [101], it is possible for the pursuer to win. However, the computation is $\mathcal{O}(m^{kd})$ where k is the number of players and d the dimensions.

As the number of players grows, computing the value function becomes infeasible. We propose locally optimal strategies for the multiplayer surveillance-evasion game, based on the value function for the static game. In addition, we propose a deep neural network trained via self play and Monte Carlo tree search to learn controls for the pursuer. Unlike Go, Chess,

and Shogi, the surveillance-evasion game is not symmetric; the pursuers and evaders require different tactics. We use the local strategies to help improve the efficiency of self-play.

The neural network is trained offline on a class of environments. Then, during play time, the trained network can be used to play games efficiently on previously unseen environments. That is, at the expense of preprocessing time and optimality, we present an algorithm which can run efficiently. While the deviation from optimality may sound undesirable, it actually is reasonable. Optimality assumes perfect actions and instant reactions. It real applications, noise and delays will perturb the system away from optimal trajectories. We show promising examples in 2D.

3.2 Value function from HJI equation

Without loss of generality, we formulate the two player game, with a single pursuer and single evader. The domain $\Omega \subseteq \mathbb{R}^d$ consists of obstacles and free space: $\Omega = \Omega_{\text{obs}} \cup \Omega_{\text{free}}$. Consider a pursuer and evader whose positions at a particular time instance are given by $P, E : [0, \infty) \to \Omega_{\text{free}}$, respectively. Let $A := S^{d-1} \cup \{\mathbf{0}\}$ be the compact set of control values. The feedback controls map the players' positions to a control value:

$$\sigma_P, \sigma_E \in \mathcal{A} := \{\sigma : \Omega_{\text{free}} \times \Omega_{\text{free}} \to A \mid \sigma \text{ measurable}\}, \tag{3.1}$$

where $\mathcal{A}$ is the set of admissible controls. The players move with velocities $f_P, f_E : \Omega \to [0, \infty)$ according to the dynamics

$$\begin{aligned} \dot{P}(t) &= f_P(P(t))\sigma_P(P(t), E(t)) & \dot{E}(t) &= f_E(E(t))\sigma_E(P(t), E(t)) \\ P(0) &= P^0 & E(0) &= E^0 \end{aligned} \tag{3.2}$$

For clarity of notation, we will omit the dependence of the controls on the players' positions. For simplicity, we assume velocities are isotropic, meaning they do not depend on the controls. In real-world scenarios, this may not be the case. For example, an airplane's dynamics might be constrained by its momentum and turning radius.

As a slight relaxation, we consider the finite-horizon version of the game, where the pursuers win if they can prolong the game past a time threshold T. Let $\mathcal{T}_{\text{end}} := \{(P(\cdot), E(\cdot))|\xi(P(\cdot), E(\cdot)) \leq 0\}$ be the end-game set of losing positions, where ξ is the shadow function defined in section 1.2. Define the payoff function

$$\mathcal{J}[P^0, E^0, t, \sigma_P, \sigma_E] := \inf\{0 \leq \tau \leq t|(P(\tau), E(\tau)) \in \mathcal{T}_{\text{end}}\}, \tag{3.3}$$

where $\mathcal{J}[P^0, E^0, t, \sigma_P, \sigma_E] := t$ if the set $(P(\tau), E(\tau)) \in \mathcal{T}_{\text{end}}$ is empty. The payoff is the minimum time-to-occlusion for given set initial positions and controls. Define the finite-horizon value function as:

$$V(P^0, E^0, t) = \sup_{\sigma_P \in \mathcal{A}} \inf_{\sigma_E \in \mathcal{A}} \mathcal{J}[P^0, E^0, t, \sigma_P, \sigma_E] \tag{3.4}$$

The value function describes the length of the game played to time t, starting from all pairs of initial positions, and assuming optimal controls. We are interested in $V(P^0, E^0, T)$ for a sufficiently large T, which characterizes the

set of initial positions from which the pursuers can maintain visibility of the evaders for at least T time units. As $T \to \infty$, we recover the infinite-horizon value function.

By using the principle of optimality and Taylor expansion, one can derive the Hamilton-Jacobi-Isaacs equation [4, 23, 26]:

$$\begin{aligned} V_t + \inf_{\sigma_E \in A} \sup_{\sigma_P \in A} \{-\nabla_P V \cdot \sigma_P - \nabla_E V \cdot \sigma_E\} = 1 \ , \quad & \text{on } \Omega_{\text{free}} \setminus \mathcal{T}_{\text{end}} \\ V(P, E, 0) = 0 & \\ V(P, E, t) = 0 \ , \quad (P, E) \in \mathcal{T}_{\text{end}} & \\ V(P, E, t) = \infty \ , \quad P \text{ or } E \in \Omega_{\text{obs}} & \end{aligned} \tag{3.5}$$

It has been shown the value function is the viscosity solution [4, 23, 26] to (3.5). For isotropic controls, this simplifies to the following Eikonal equation:

$$\begin{aligned} V_t - f_P |\nabla_P V| + f_E |\nabla_E V| = 1 \ , \quad & \text{on } \Omega_{\text{free}} \setminus \mathcal{T}_{\text{end}} \\ V(P, E, 0) = 0 & \\ V(P, E, t) = 0 \ , \quad (P, E) \in \mathcal{T}_{\text{end}} & \\ V(P, E, t) = \infty \ , \quad P \text{ or } E \in \Omega_{\text{obs}} & \end{aligned} \tag{3.6}$$

The optimal controls can be recovered by computing the gradient of the value function:

$$\sigma_P = \frac{\nabla_P V}{|\nabla_P V|} \ , \qquad \sigma_E = -\frac{\nabla_E V}{|\nabla_E V|} \ . \tag{3.7}$$

3.2.1 Algorithm

Following the ideas in [107], we discretize the gradient using upwind scheme as follows. Let $P_{i,j}, E_{k,l} \in \Omega_{\text{free}}$ be the discretized positions with grid

spacing h. Denote $V_{i,j,k,l}$ as the numerical solution to (3.6) for initial positions $P_{i,j}, E_{k,l}$. We estimate the gradient using finite difference. For clarity, we will only mark the relevant subscripts, e.g. $V_{i+1} := V_{i+1,j,k,l}$.

$$\begin{aligned} P_{x^-} &:= \frac{1}{h}\Big(V_i - V_{i-1}\Big), & E_{x^-} &:= \frac{1}{h}\Big(V_k - V_{k-1}\Big), \\ P_{x^+} &:= \frac{1}{h}\Big(V_{i+1} - V_i\Big), & E_{x^+} &:= \frac{1}{h}\Big(V_{k+1} - V_k\Big), \\ P_{y^-} &:= \frac{1}{h}\Big(V_j - V_{j-1}\Big), & E_{y^-} &:= \frac{1}{h}\Big(V_l - V_{l-1}\Big), \\ P_{y^+} &:= \frac{1}{h}\Big(V_{j+1} - V_j\Big), & E_{y^+} &:= \frac{1}{h}\Big(V_{l+1} - V_l\Big). \end{aligned} \tag{3.8}$$

Let $a^- := -\min(0, a)$ and $a^+ := \max(0, a)$. Define

$$\operatorname{sgn}\max(a, b) := \begin{cases} a^+ & \text{if } \max(a^+, b^-) = a^+ \\ -b^- & \text{if } \max(a^+, b^-) = b^- \end{cases} \tag{3.9}$$

and

$$\begin{aligned} \partial P_x V &= \operatorname{sgn}\max(P_{x^+}, P_{x^-}) \\ \partial P_y V &= \operatorname{sgn}\max(P_{y^+}, P_{y^-}) \\ \partial E_x V &= \operatorname{sgn}\max(E_{x^-}, E_{x^+}) \\ \partial E_y V &= \operatorname{sgn}\max(E_{y^-}, E_{y^+}) \end{aligned} \tag{3.10}$$

Finally, the desired numerical gradients are

$$\begin{aligned} |\nabla_P V| &= \Big((\partial P_x V)^2 + (\partial P_y V)^2\Big)^{1/2} \\ |\nabla_E V| &= \Big((\partial E_x V)^2 + (\partial E_y V)^2\Big)^{1/2} \end{aligned} \tag{3.11}$$

Then we have a simple explicit scheme.

$$V^{n+1} = V^n + \Delta t(1 + f_P|\nabla_P V| - f_E|\nabla_E V|) \tag{3.12}$$

The CFL conditions dictate that the time step Δt should be

$$\Delta t \leq \frac{h}{16 \max(f_P, f_E)} \tag{3.13}$$

For a given environment, we precompute the value function by iteration until convergence. During play time, we initialize P^0, E^0 and compute the optimal trajectories according to (3.7) using Δt time increments.

3.2.1.1 Boundary conditions

The obstacles appear in the HJI equation as boundary conditions. However, direct numerical implementation leads to artifacts near the obstacles. Instead, we model the obstacles by setting the velocities to be small inside obstacles. We regularize the velocities by adding a smooth transition [101]:

$$v_\epsilon(x) = \begin{cases} v(x) & \phi(x) > 0 \\ v_{\min} + \frac{v(x) - v_{\min}}{2}\left[\cos\left(\frac{\phi(x)\pi}{2\epsilon}\right) + 1\right] & \phi(x) \in [-2\epsilon, 0] \\ v_{\min} & \phi(x) < -2\epsilon \end{cases} \tag{3.14}$$

where ϕ is the signed distance function to the obstacle boundaries. In the numerical experiments, we use $\epsilon = 16\Delta x$ and $v_{\min} = 1/100$.

3.2.2 Numerical results

Stationary pursuer

We verify that the scheme converges numerically for the case in which the pursuer is stationary. When $f_P = 0$, the HJI equation (3.6) becomes the

time-dependent Eikonal equation:

$$\begin{aligned} V_t + f_E|\nabla_E V| &= 1 \qquad && \text{on } \Omega_{\text{free}} \setminus \mathcal{T}_{\text{end}} \\ V(P,E,0) &= 0 \\ V(P,E,t) &= 0 \qquad && (P,E) \in \mathcal{T}_{\text{end}} \end{aligned} \tag{3.15}$$

In particular, for sufficiently large t, the value function reaches a steady state, and satisfies the Eikonal equation:

$$\begin{aligned} f_E|\nabla_E V| &= 1 \qquad && \text{on } \Omega_{\text{free}} \setminus \mathcal{T}_{\text{end}} \\ V(P,E) &= 0 \qquad && (P,E) \in \mathcal{T}_{\text{end}} \end{aligned} \tag{3.16}$$

For this special case, the exact solution is known; the solution corresponds to the evader's travel time to the end-game set $\mathcal{T}_{\text{end}}$. Also, this case effectively reduces the computational cost from $\mathcal{O}(m^4)$ to $\mathcal{O}(m^2)$, so that we can reasonably compute solutions at higher resolutions.

We consider a $\Omega = [0,1) \times [0,1)$ with a single circular obstacle of radius 0.15 centered at $(1/2, 1/2)$. The pursuer is stationary at $P_0 = (1/8, 1/2)$. We use $\Delta t = \Delta x/20$ and iterate until the solution no longer changes in the L_1 sense, using a tolerance of 10^{-5}.

We compute the "exact" solution using fast marching method on high resolution grid $M = 2048$. We vary m from 16 to 1024 and observe convergence in the L_1 and L_2 sense, as seen in Table 3.1. In Figure 3.1, we plot the level curves comparing the computed solution at $m = 512, 1024$ to the "exact" solution. Notice the discrepancies are a result of the difficulty in dealing with boundary conditions. However, these errors decay as the grid is refined.

The case where the evader is stationary is not interesting.

Table 3.1: Error for the stationary pursuer case, compared to the known solution computed using fast marching method at resolution $M = 2048$.

m	L_1 error	L_2 error
16	0.08972215	0.01288563
32	0.03177683	0.00159669
64	0.02442984	0.00111537
128	0.01059728	0.00021345
256	0.00515584	0.00005214
512	0.00304322	0.00001961
1024	0.00086068	0.00000142

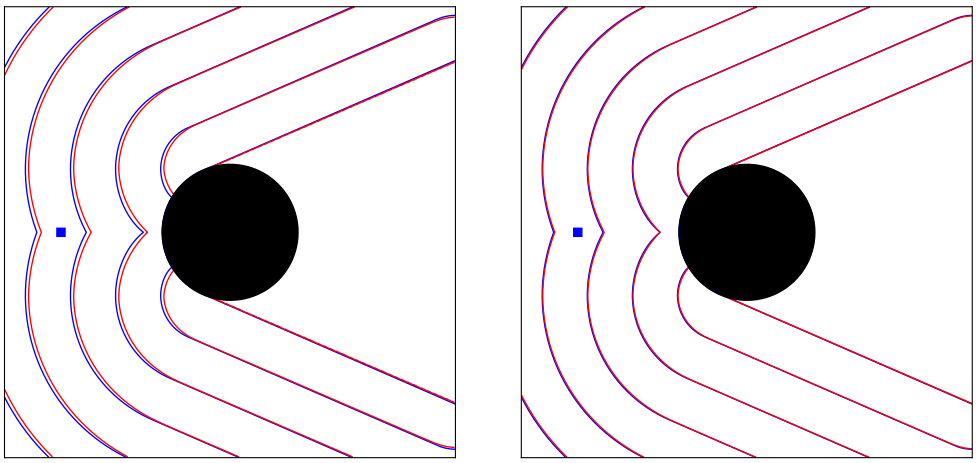

Figure 3.1: Comparison of contours of the "exact" solution (blue) with those computed by the scheme (3.12) (red) using grid resolutions $m = 512$ (left) and $m = 1024$ (right). The pursuer (blue square) is stationary. The error emanates from the obstacle due to boundary conditions, but the scheme converges as the grid is refined.

A circular obstacle

The evader has the advantage in the surveillance-evasion game. It is difficult for the pursuer to win unless it is sufficiently fast. But once it is fast enough, it can almost always win. Define the winning regions for the pursuer and evader, respectively:

$$\mathcal{W}_P = \{(P, E) | V(P, E) > T_{\max}\} \tag{3.17}$$

$$\mathcal{W}_E = \{(P, E) | V(P, E) \leq T_{\max}\} \tag{3.18}$$

Here, we use $T_{\max} = .9T$ to tolerate numerical artifacts due to boundary conditions. In Figure 3.2, we show how the winning region for a fixed evader/pursuer position changes as the pursuer's speed increases. Since it is difficult to visualize data in 4D, we plot the slices $V(P^0, \cdot)$ and $V(\cdot, E^0)$ where $P^0 = E^0 = (1/8, 1/2)$. We use $m = 64$, $\Delta t = \Delta x/20$ and iterate until $T = 10$. We fix $f_P = 1$ and compute V for each $f_E \in \{\frac{1}{3}, \frac{1}{2}, \frac{2}{3}\}$. The computation for each value function takes 16 hours.

Figure 3.3 shows trajectories from several initial positions with various speeds. Interestingly, once the evader is cornered, the optimal controls dictate that it is futile to move. That is, the value function is locally constant.

More obstacles

In Figure 3.4 we consider a more complicated environment with multiple obstacles. Here, the pursuer is twice as fast as the evader. Although there are many obstacles, the dynamics are not so interesting in the sense that the

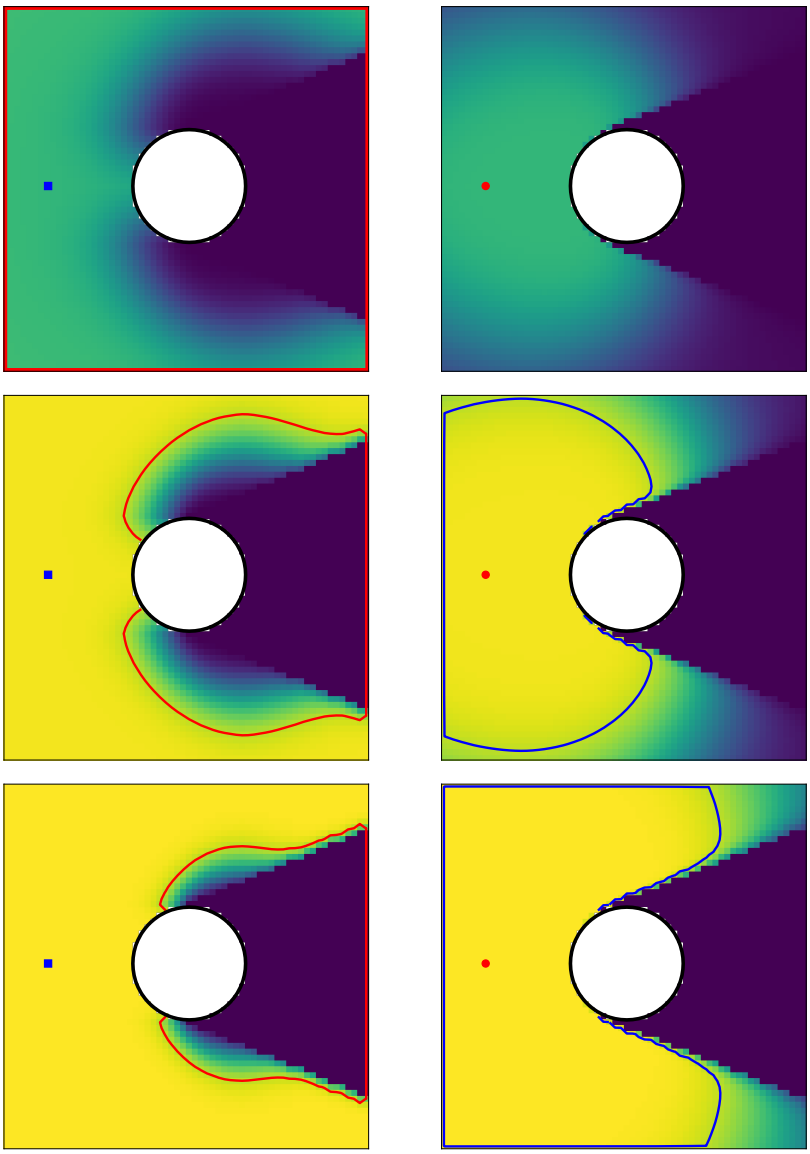

Figure 3.2: Comparison of winning initial positions for the evader (left, red contour) against a pursuer with fixed initial position (blue square) and vice versa – winning initial positions for the pursuer (right, blue contour) against an evader with fixed initial position (red circle). Left column shows $V(P^0, \cdot)$ while right column shows $V(\cdot, E^0)$, where higher values of V are yellow, while lower values are dark blue. From top to bottom, the pursuer is 1.5, 2 and 3 times faster than the evader. The pursuer must be sufficiently fast to have a chance at winning.

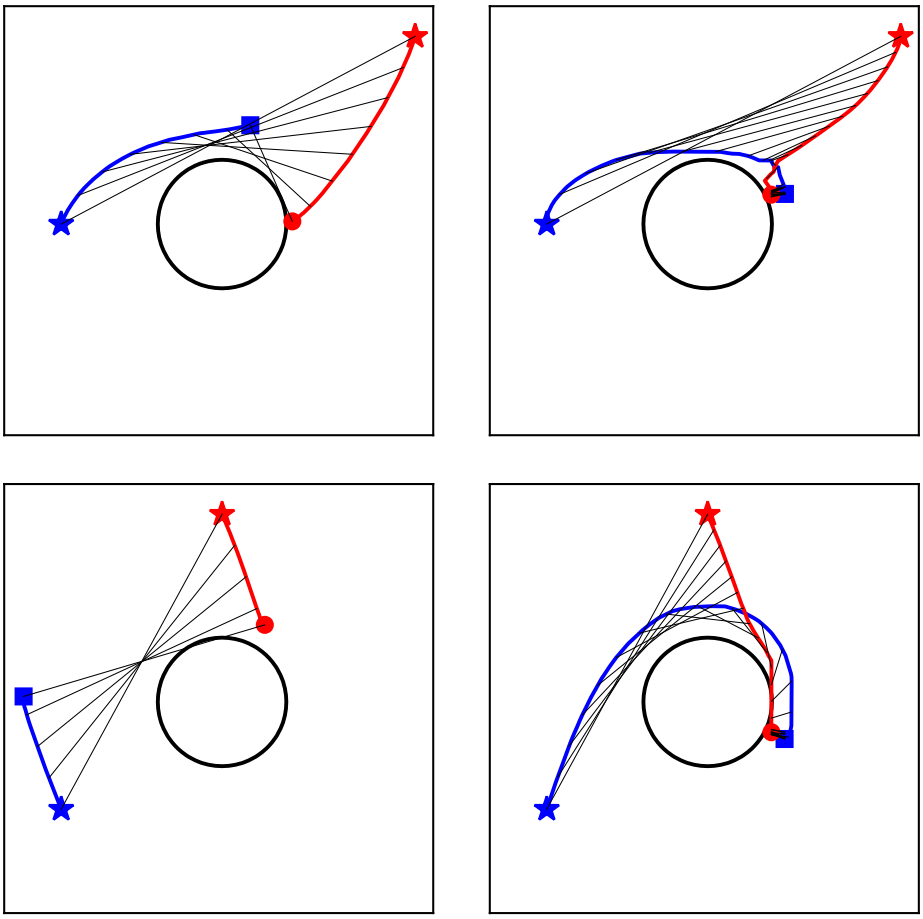

Figure 3.3: Trajectories of several games played around a circle. The pursuer loses when it has same speed as the evader (left column). When the pursuer is 2x faster than the evader, it is possible to win; the evader essentially gives up once it is cornered, since no controls will change the outcome (right column). Initial positions are shown as stars. Black lines connect positions at constant time intervals.

evader will generally navigate towards a single obstacle. Again, the evader tends to give up once the game has been decided.

Finally, in Figure 3.5 we show suboptimal controls for the evader. In particular, the evader is controlled manually. Although manual controls do not help the evader win, they lead to more interesting trajectories.

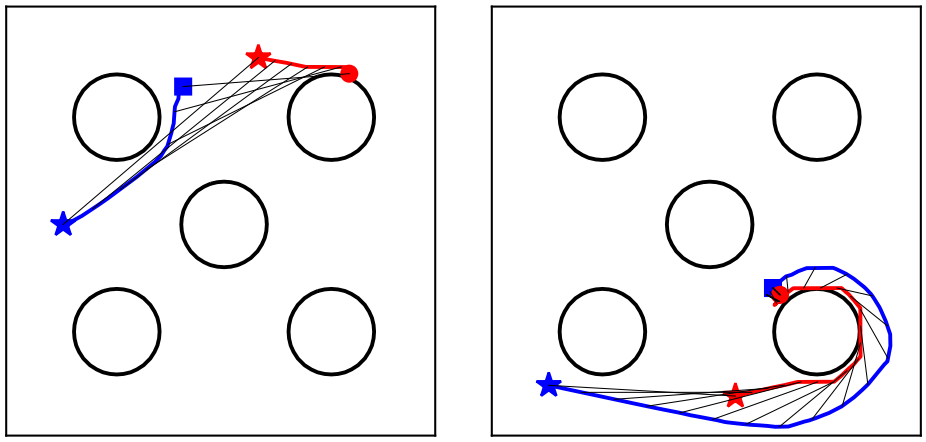

Figure 3.4: Trajectories of games played around 5 circular obstacles. Pursuer (blue) is 2x as fast as evader (red). The evader wins (left) if it can quickly hide. Otherwise it will give up once it is captured (right). Initial positions are shown as stars. Black lines connect positions at constant time intervals.

3.2.3 Discussion

Notice that an optimal trajectory for the pursuer balances distance and visibility. While moving closer to the evader will guarantee that it can't "get away", it is not sufficient, since being close leads to more occlussions. On the other hand, moving far away gives better visibility of the environment, but may make it impossible to catch up once E turns the corner.

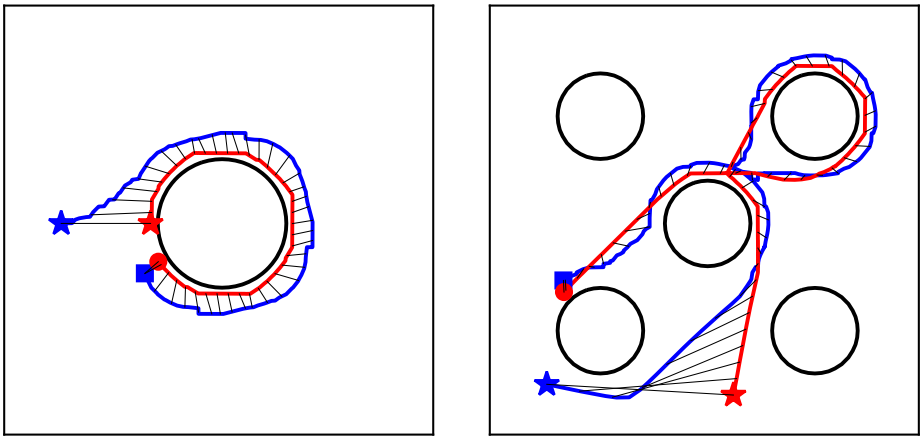

Figure 3.5: Manually controlled evader against an optimal pursuer. The evader loses in both cases, but does not give up.

Although we described the formulation for a single pursuer and evader, the same scheme holds for multiple pursuers and evaders. The end-game set just needs to be modified to take into account the multiple players. That is, the game ends as soon as any evader is occluded from all pursuers. However, the computational complexity of the scheme is $\mathcal{O}(m^{kd})$ which quickly becomes unfeasible even on small grid sizes. At the expense of offline compute time, the game can be played efficiently online. The caveat is that, the value function is only valid for the specific map and velocities computed offline.

Note that in the limit of infinite evaders, the game reduces to the surveillance problem from chapter 2.

3.3 Locally optimal strategies

As the number of players increases, computing the value function from the HJI equations is no longer tractable. We consider a discrete version of the game, with the aim of feasibly computing controls for games with multiple pursuers and multiple evaders. Each player's position is now restricted on a grid, and at each turn, the player can move to a new position within a neighborhood determined by its velocity. Players cannot move through obstacles. Formally, define the arrival time function

$$\begin{aligned} d_P(x,y) &:= \min_{\sigma_P \in \mathcal{A}} \min\{t | P(0) = x, P(t) = y\} \\ d_E(x,y) &:= \min_{\sigma_E \in \mathcal{A}} \min\{t | E(0) = x, E(t) = y\}. \end{aligned} \tag{3.19}$$

The set of valid actions are the positions y which can be reached from x within a Δt time increment:

$$\begin{aligned} A_P(t) &:= \{y \in \Omega_{\text{free}} | d_P(P(t), y) \leq \Delta t\} \\ A_E(t) &:= \{y \in \Omega_{\text{free}} | d_E(E(t), y) \leq \Delta t\}. \end{aligned} \tag{3.20}$$

In a Δt time step, each player can move to a position

$$\begin{aligned} P(t + \Delta t) &\in A_P(t) \\ E(t + \Delta t) &\in A_E(t). \end{aligned} \tag{3.21}$$

Analogously, for multiple players, denote the number of pursuers and evaders as k_P and k_E, respectively. Define

$$\begin{aligned} \mathbf{P} &= (P_1, \ldots, P_{k_P}) \\ \mathbf{E} &= (E_1, \ldots, E_{k_E}) \\ \mathbf{A_P}(t) &= \{\mathbf{P} | P_i \in A_P(t) \ , \ i = 1, \ldots, k_P\} \\ \mathbf{A_E}(t) &= \{\mathbf{E} | E_j \in A_E(t) \ , \ j = 1, \ldots, k_E\} \end{aligned} \tag{3.22}$$

so that in Δt time, each team can move to

$$\begin{aligned} \mathbf{P}(t+\Delta t) &\in \mathbf{A_P}(t) \\ \mathbf{E}(t+\Delta t) &\in \mathbf{A_E}(t) \end{aligned} \tag{3.23}$$

The game ends as soon as one evader is occluded from all pursuers. The end-game set is

$$\mathcal{T}_{\text{end}} = \{(\mathbf{P}, \mathbf{E}) \mid \exists\ j : \xi(P_i, E_j) \leq 0 \text{ for } i = 1, \ldots, k_P\}, \tag{3.24}$$

We propose two locally optimal strategies for the pursuer.

3.3.1 Distance strategy

The trajectories from the section 3.2 suggest that the pursuer must generally remain close to the evader. Otherwise, the evader can quickly hide behind obstacles. A simple strategy for the pursuer is to move towards the evader:

$$P(t+\Delta t) = \operatorname*{arg\,min}_{x \in A_P(t)} d_P\Big(x, E(t)\Big). \tag{3.25}$$

That is, in the time increment Δt, the pursuer should pick the action that minimizes its travel time to the evader's current position at time t.

For the multiplayer game, we propose a variant of the Hausdorff distance, where max is replaced by a sum:

$$\mathbf{d_P}(\mathbf{x}, \mathbf{E}(t)) := \frac{1}{2}\Big[\sum_{i=1}^{k_P} \min_j d_{P_i}\Big(x_i, E_j(t)\Big)^2\ \Big]^{1/2} + \frac{1}{2}\Big[\sum_{j=1}^{k_E} \min_i d_{P_i}\Big(x_i, E_j(t)\Big)^2\ \Big]^{1/2}.$$

Informally, the first term encourages each pursuer to be close to an evader, while the second term encourages a pursuer to be close to each evader. The sum

helps to prevent ties. The optimal action according to the distance strategy is

$$\mathbf{P}(t+\Delta t) = \underset{\mathbf{x}\in\mathbf{A_P}(t)}{\arg\min}\, \mathbf{d_P}(\mathbf{x}, \mathbf{E}(t)) \tag{3.26}$$

In the next section, we will use search algorithms to refine policies. Rather than determining the *best* action, it is useful to quantify the utility of each action. To do so, define $p_{\text{distance}} : \Omega \times \Omega_{\text{free}} \times \cdots \times \Omega_{\text{free}} \to \mathbb{R}$ as the policy, which outputs a probability the agent should take an action, conditioned on the current player positions. We normalize using the *softmax* function to generate the policy:

$$\begin{aligned} \alpha &= \Big[\sum_{\mathbf{x}\in\mathbf{A_P}(t)} e^{-\mathbf{d_P}(\mathbf{x},\mathbf{E}(t))} \Big]^{-1} \\ p_{\text{distance}}\big(\mathbf{x}|(\mathbf{P}(t),\mathbf{E}(t))\big) &= \begin{cases} \alpha e^{-\mathbf{d_P}(\mathbf{x},\mathbf{E}(t))} & \mathbf{x}\in\mathbf{A_P}(t) \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{3.27}$$

For the discrete game with Δt time increments, one can enumerate the possible positions for each P_i and evaluate the travel time to find the optimal control. The arrival time function $d_{P_i}(\cdot, E_j(t))$ to each E_j at the current time t can be precomputed in $\mathcal{O}(m^d)$ time. In the general case, where each pursuer may have a different velocity field f_{Pi}, one would need to compute k_P arrival time functions. If $a_{\Delta t}(P_i)$ is the max number of possible actions each pursuer can make in Δt increment, then the total computational complexity for one move is

$$O\Big(k_P k_E m^d + \prod_{i=1}^{k_P} a_{\Delta_t}(P_i)\Big).$$

For the special case when the pursuers have the same f_P, the complexity reduces to

$$O\Big(k_E m^d + \prod_{i=1}^{k_P} a_{\Delta_t}(P_i)\Big).$$

3.3.2 Shadow strategy

Recall that, for a stationary pursuer, the value function for the evader becomes the Eikonal equation:

$$\begin{aligned} f_E|\nabla_E V| &= 1 && \text{on } \Omega_{\text{free}} \setminus \mathcal{T}_{\text{end}} \\ V(P,E) &= 0 && (P,E) \in \mathcal{T}_{\text{end}}, \end{aligned} \tag{3.28}$$

whose solution is the travel time to the shadow set. Define the time-to-occlusion as

$$\tau_E(P, E^0) := \min_{\sigma_E \in \mathcal{A}} \min\{t \geq 0 \mid E(0) = E^0 \ , \ (P, E(t)) \in \mathcal{T}_{\text{end}} \ \}. \tag{3.29}$$

It is the shortest time in which an evader at E^0 can be occluded from a stationary pursuer at P. Thus, a reasonable strategy for the evader is to pick the action which brings it closest to the shadow formed by the pursuer's position:

$$E(t + \Delta t) = \arg\min_{y \in A_E(t)} \tau_E(P(t), y). \tag{3.30}$$

A conservative strategy for the pursuer, then, is to maximize time-to-occlusion, assuming that the evader can anticipate its actions:

$$\tau_E^*(x, E(t)) = \min_{y \in A_E(t)} \tau_E(x, y) \tag{3.31}$$

$$P(t + \Delta t) = \arg\max_{x \in A_P(t)} \tau_E^*(x). \tag{3.32}$$

Remark: The strategy (3.32) is a local variant of the static value function proposed in [101]. In that paper, they suggest using the static value function for feedback controls by moving towards the globally optimal destination, and then recomputing at Δt time intervals. Here, we use the locally optimal action.

For multiple players, the game ends as soon as any evader is hidden from all pursuers. Define the time-to-occlusion for multiple players:

$$\boldsymbol{\tau}_{\mathbf{E}}(\mathbf{P}, \mathbf{E}^0) := \min_{\sigma_{Ei} \in \mathcal{A}} \min\{t \geq 0 \mid \mathbf{E}(0) = \mathbf{E}^0 \ , \ (\mathbf{P}, \mathbf{E}(t)) \in \mathcal{T}_{\text{end}} \ \}. \tag{3.33}$$

Then, the strategy should consider the shortest time-to-occlusion among all possible evaders' actions in the Δt time increment:

$$\begin{aligned} \boldsymbol{\tau}_{\mathbf{E}}{}^*(\mathbf{x}, \mathbf{E}(t)) &:= \min_{\mathbf{y} \in \mathbf{A}_{\mathbf{E}}(t)} \boldsymbol{\tau}_{\mathbf{E}}(\mathbf{x}, \mathbf{y}) \\ \mathbf{P}(t + \Delta t) &= \arg\max_{\mathbf{x} \in \mathbf{A}_{\mathbf{P}}(t)} \boldsymbol{\tau}_{\mathbf{E}}{}^*(\mathbf{x}, \mathbf{E}(t)). \end{aligned} \tag{3.34}$$

The corresponding shadow policy is:

$$\begin{aligned} \alpha &= \Big[\sum_{x \in \mathbf{A}_{\mathbf{P}}(t)} e^{\boldsymbol{\tau}_{\mathbf{E}}{}^*(\mathbf{x}, \mathbf{E}(t))} \Big]^{-1} \\ p_{\text{shadow}}\big(\mathbf{x} | (\mathbf{P}(t), \mathbf{E}(t))\big) &= \begin{cases} \alpha e^{\boldsymbol{\tau}_{\mathbf{E}}{}^*(\mathbf{x}, \mathbf{E}(t))} & \mathbf{x} \in \mathbf{A}_{\mathbf{P}}(t) \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{3.35}$$

This strategy is computationally expensive. One can precompute the arrival time to each evader by solving an Eikonal equation $\mathcal{O}(m^d)$. For each combination of pursuer positions, one must compute the joint visibility function and corresponding shadow function $\mathcal{O}(m^d)$. Then the time-to-occlusion

can be found by evaluating the precomputed arrival times to find the minimum within the shadow set $\mathcal{O}(m^d)$. The computational complexity for one move is

$$O\Big(k_E m^d + m^d \cdot \prod_{i=1}^{k_P} a_{\Delta t}(P_i)\Big). \tag{3.36}$$

One may also consider alternating minimization strategies to achieve

$$O\Big(k_E m^d + m^d \cdot \sum_{i=1}^{k_P} a_{\Delta t}(P_i)\Big), \tag{3.37}$$

though we leave that for future work.

Blend strategy

We have seen from 3.2 that optimal controls for the pursuer balance the distance to, and visibility of, the evader. Thus a reasonable approach would be to combine the distance and shadow strategies. However, it is not clear how they should be integrated. One may consider a linear combination, but the appropriate weighting depends on the game settings and environment. Empirically, we observe that the product of the policies provides promising results across a range of scenarios. Specifically,

$$p_{\text{blend}} \propto p_{\text{shadow}} \cdot p_{\text{distance}} \tag{3.38}$$

3.3.3 Numerical results

We present some representative examples of the local policies. First, we consider the game with a circular obstacle, a single purser and single evader whose speeds are $f_P = 3$ and $f_E = 2$, respectively. Figure 3.6 illustrates the

typical trajectories for each policy. In general, the distance strategy leads the pursuer into a cat-and-mouse game the with evader; the pursuer, when close enough, will jump to the evader's position at the previous time step. The shadow strategy keeps the pursuer far away from obstacles, since this allows it to *steer the shadows* in the fastest way. The blend strategy balances the two approaches and resembles the optimal trajectories based on the HJI equation in section 3.2.

Next, we highlight the advantages of the shadow strategy with a 2 pursuer, 2 evader game on a map with two crescent-shaped obstacles. The pursuer and evader speeds are $f_P = 4$ and $f_E = 2$, respectively. The openness of the environment creates large occlusions. The pursuers use the shadow strategy to cooperate and essentially corner the evaders. Figure 3.7 shows snapshots of the game. The distance strategy loses immediately since the green purser does not properly track the orange evader.

We present cases where the distance and shadow strategy fail in Figure 3.8. The evader tends to stay close to the obstacle, since that enables the shortest path around the obstacle. Using the distance strategy, the pursuer agressively follows the evader. The evader is able to counter by quickly jumping behind sharp corners. On the other hand, the shadow strategy moves the pursuer away from obstacles to reduce the size of shadows. As a consequence, the pursuer will generally be too far away from the evader and eventually lose. In environments with many nonconvex obstacles, both strategies will fail.

Finally, we show that blending the shadow and distance strategies is

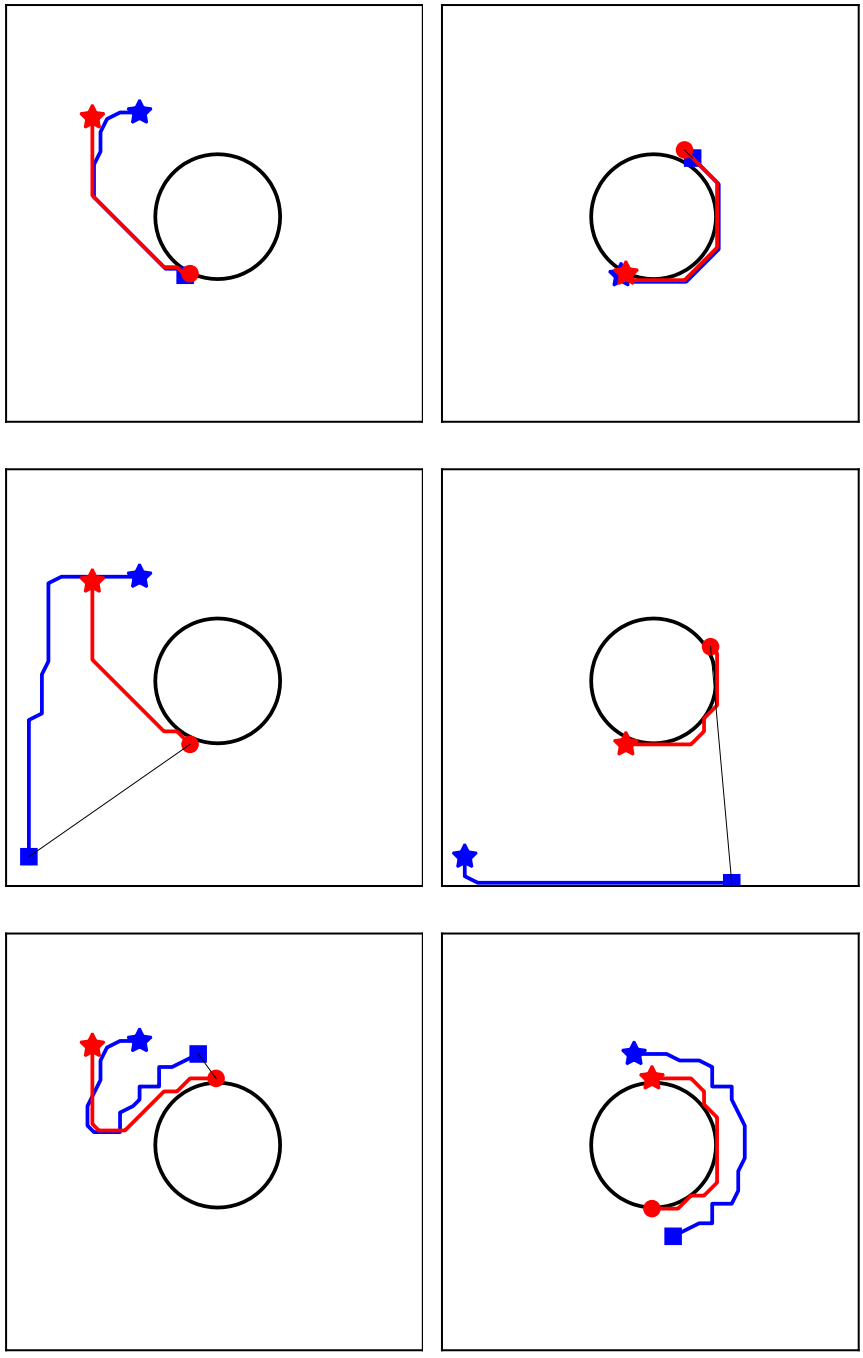

Figure 3.6: Distance strategy (top) follows the evader closely, shadow strategy (middle) stays far to gain better perspective, while the blend strategy (bottom) strikes a balance.

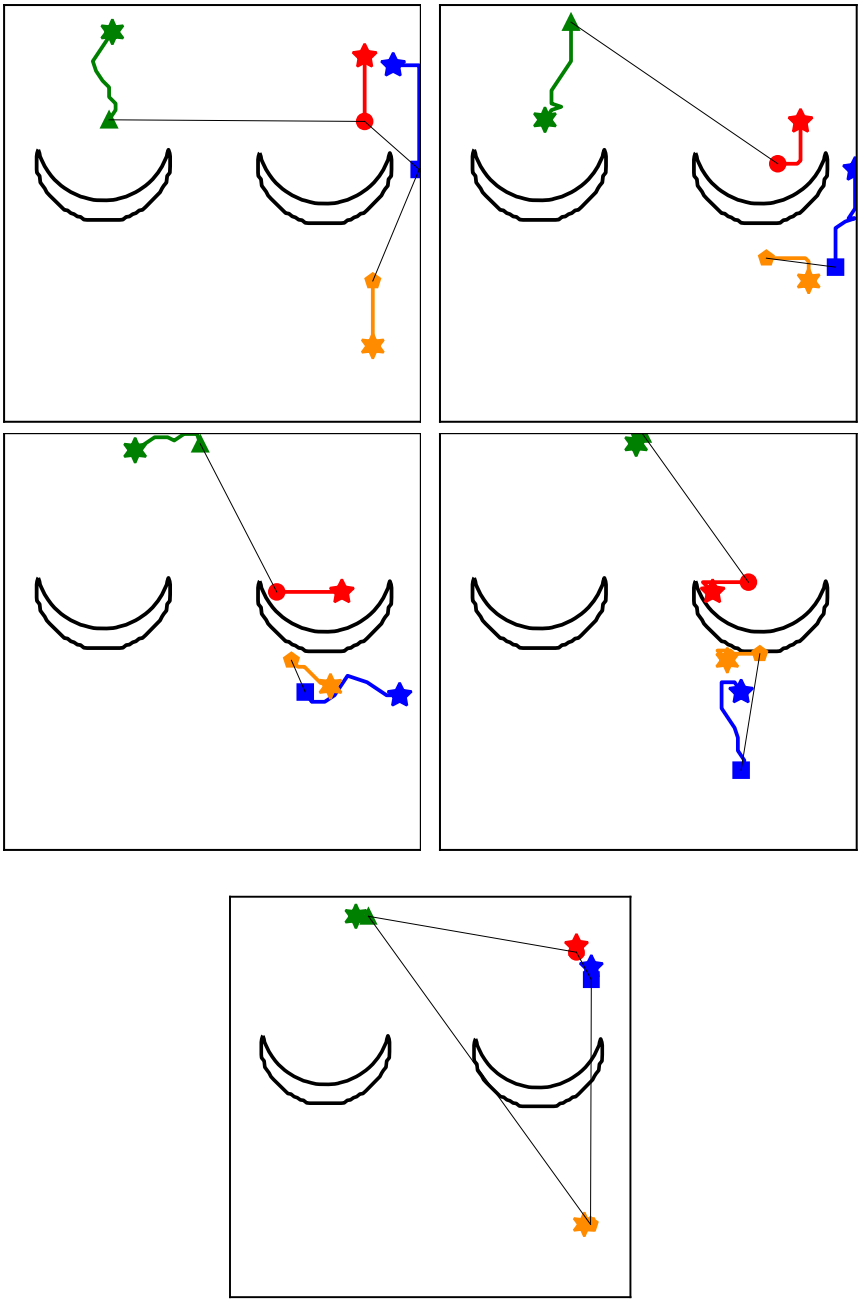

Figure 3.7: (Top 2 rows) The blue and green pursuers cooperate by using the shadow strategy. Green initially has responsibility of the orange evader, but blue is able to take over. (Bottom) The distance strategy loses immediately.

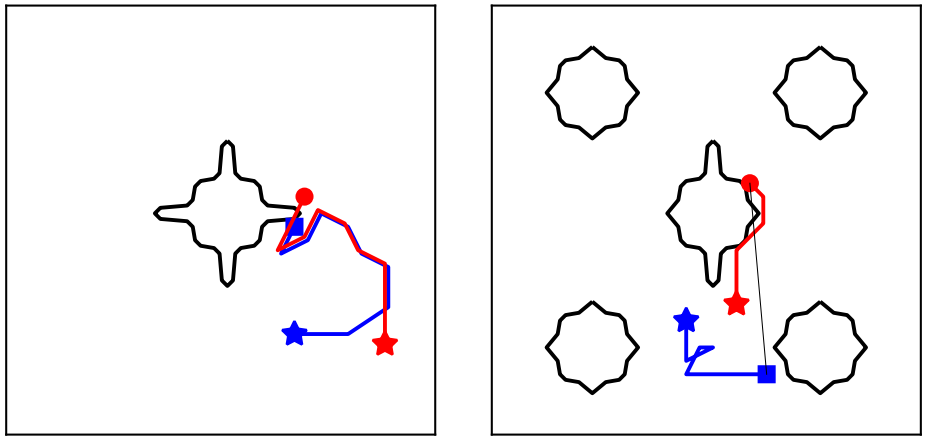

Figure 3.8: Failure modes for the local strategies. Blindly using the distance strategy (left) allows the evader to exploit the sharp concavities. The shadow strategy (right) keeps the pursuer far away to reduce the size of shadows, but often, the pursuer is too far away to catch the evader.

very effective in compensating for the shortcomings of each individual policy. The pursuers are able to efficiently track the evaders while maintain a safe distance. Figure 3.9 shows an example with 2 pursuers and 2 evaders on a map with multiple obstacles, where $f_P = 3$ and $f_E = 2$.

3.4 Learning the pursuer policy

We propose a method for learning optimal controls for the pursuer, though our methods can be applied to find controls for the evader as well. Again, we consider a discrete game, where each player's position is restricted on a grid, and at each turn, the player can move to a new position within a neighborhood determined by their velocity. All players move simultaneously.

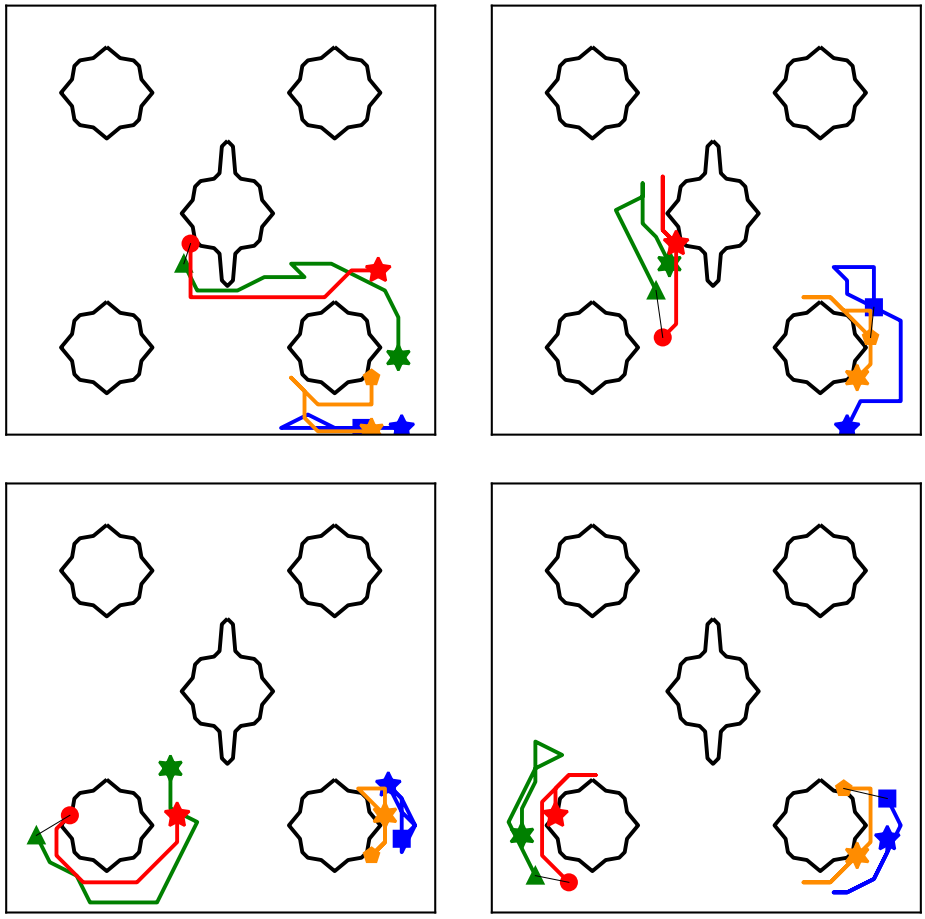

Figure 3.9: The pursuers (blue and green) are able to win by combining the distance and shadow strategy. The pursuers stay close, while maintaining enough distance to avoid creating large shadow regions. The pursuers are slightly faster than the evaders.

The game termination conditions are checked at the end of each turn.

We initialize a neural network which takes as input any game state, and produces a policy and value pair. The policy is probability distribution over actions. Unlike in section 3.2, the value, in this context, is an estimate of the likely winner given the input state.

Initially the policy and value estimates are random. We use Monte Carlo tree search to compute refined policies. We play the game using the refined policies, and train the neural network to learn the refined policies. By iterating in this feedback loop, the neural network continually learns to improve its policy and value estimates. We train on games in various environments and play games on-line on maps that were were not seen during the training phase.

3.4.1 Monte Carlo tree search

In this section, we review the Monte Carlo tree search algorithm, which allows the agent to plan ahead and refine policies. For clarity of notation, we describe the single pursuer, single evader scenario, but the method applies to arbitrary number of players.

Define the set of game states $\mathcal{S} := \{(P, E) \in \Omega_{\text{free}} \times \Omega_{\text{free}}\}$ so that each state $s \in \mathcal{S}$ characterizes the position of the players. Let $\mathcal{A} \subseteq \Omega_{\text{free}}$ be the set of actions. Let $T(s, a) : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ be the transition function which outputs the state resulting from taking action a at state s. Let $f : \mathcal{S} \to \mathbb{R}^{m^d} \times [-1, 1]$ be an evaluator function which takes the current state as input and provides

a policy and value estimate: $f(s) = (\vec{p}, v)$. Formally, Monte Carlo tree search is mapping takes as input the current state s_0, the evaluator function f, and a parameter M indicating the number of search iterations: MCTS$(s_0, f; M)$. It outputs a refined policy $\vec{\pi}^*$.

Algorithm 3.1 summarizes the MCTS algorithm. At a high level, MCTS simulates game play starting from the current state, keeping track of nodes it has visited during the search. Each action is chosen according to a formula $U(s, a)$ which balances exploration and exploitation. Simulation continues until the algorithm reaches a *leaf node* s_n, a state which has not previously been visited. At this point, we use the evaluator function $f(s_n) = (\vec{p}, v)$ to estimate a policy and value for that leaf node. The value v is propagated to all parent nodes. One iteration of MCTS ends when it reaches a leaf node. MCTS keeps track of statistics that help guide the search. In particular

- $N(s, a)$: the number of times the action a has been selected from state s
- $W(s, a)$: the cumulative value estimate for each state-action pair
- $Q(s, a)$: the mean value estimate for each state-action pair
- $P(s, a) = (1 - \varepsilon)p(a|s) + \varepsilon\eta$: the prior policy, computed by evaluating f. Dirichlet noise η is added to allow a chance for each move to be chosen.
- $U(s, a) = Q(s, a) + P(s, a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$ is the *upper confidence bound* [80]. The first term exploits moves with high value, while the second term encourages moves that have not selected.

When all M iterations are completed, the desired refined policy is proportional to $N(s_0, a)^{1/\tau}$, where τ is a smoothing term.

Algorithm 3.1 Monte Carlo tree search: MCTS(s_0, f, M)

$N(s, a) \leftarrow 0$
$Q(s, a) \leftarrow 0$
$W(s, a) \leftarrow 0$
visited $= \{\emptyset\}$
for $i = 1, \ldots, M$ **do**
  $n \leftarrow 0$
  while $s_n \notin$ visited **do**

    if $\sum_b N(s_n, b) > 0$ **then**
      $a_n^* = \arg\max_a Q(s_n, a) + P(s_n, a)\frac{\sqrt{\sum_b N(s_n, b)}}{1+N(s_n, a)}$
    else
      $a_n^* = \arg\max_a P(s_n, a)$
    end if
    $s_{n+1} = T(s_n, a_n^*)$
    $n \leftarrow n + 1$
  end while
  $(p, v) = f(s_n)$
  $P(s_n, a) = (1 - \varepsilon)p(a|s_n) + \varepsilon\eta$
  visited.append(s_n)
  for $j = 0, \ldots, n - 1$ **do**
    $N(s_j, a) \leftarrow N(s_j, a_j^*) + 1$
    $W(s_j, a) \leftarrow W(s_j, a_j^*) + v$
    $Q(s_j, a) \leftarrow Q(s_j, a_j^*)/N(s_j, a_j^*)$
  end for
end for
$\pi^*(a|s_0) = N(s_0, a)^{1/\tau} / \sum_b N(s_0, b)^{1/\tau}$
return π^*

3.4.2 Policy and value network

We use a convolutional neural network which takes in the game state and produces a policy and value estimate. Although the state can be completely characterized by the positions of the players and the obstacles, the neural network requires more context in order to be able to generalize to new environments. We provide the following features as input to the neural network, each of which is an $m \times m$ image:

- Obstacles as binary image
- Player positions, a separate binary image for each player
- Joint visibility of all pursuers, as a binary image
- Joint shadow boundaries of all pursuers
- Visibility from each pursuer's perspective, as a binary image
- Shadow boundary from each pursuer's perspective
- Valid actions for each player, as a binary image
- Each evader's policy according to (3.30)

AlphaZero [89] suggests that training the policy and value networks jointly improves performance. We use a single network based on U-Net [79], which splits off to give output policy and value.

The input is $m \times m \times C_{\text{in}}$, where $C_{\text{in}} = 2 + 4k_P + 2k_E$ and k_P, k_E are the number of pursuers and evaders, respectively. The U-Net consists of $\log_2(m)+1$ *down-blocks*, followed by the same number of *up-blocks*. All convolution layers in the down-blocks and up-blocks use size 3 kernels. Each down-block consists of input, conv, batch norm, relu, conv, batch norm, residual connection from input, relu, followed by downsampling with stride 2 conv, batch norm, and relu. A residual connection links the beginning and end of each block, before downsampling. The width of each conv layer in the l^{th} down-block is $l \cdot C_{\text{in}}$. Each up-block is the same as the down-block, except instead of downsampling, we use bilinear interpolation to upsample the image by a factor of 2. The upsampled result is concatenated with the predownsampled output from the corresponding (same size) down-block, followed by conv, batch norm, relu. The width of each conv layer in the up-block is same as those in the down-block of corresponding size.

Then, the network splits into a policy and value head. The policy head consists of 1×1 conv with width 8, batch norm, relu, and 1×1 conv with width k_P. The final activation layer is a softmax to output $p \in \mathbb{R}^{m \times m \times k_P}$, a policy for each pursuer. The value head is similar, with 1×1 conv with width 8, batch norm, relu, and 1×1 conv with width 1. The result passes through a tanh activation and average pooling to output a scalar $v \in [-1, 1]$.

3.4.3 Training procedure

Since we do not have the true value and policy, we cannot train the networks in the usual supervised fashion. Instead, we use MCTS to generate refined policies, which serve as the training label for the policy network. Multiple games are played with actions selected according to MCTS refined policies. The game outcomes act as the label for the value for each state in the game. We train over various maps consisting of 2-7 obstacles, including circles, ellipses, squares, and tetrahedrons.

More specifically, let $f_\theta(s)$ be the neural network parameterized by θ, which takes a state s as input, and outputs a policy $\vec{\pi}_\theta(s)$ and value $v_\theta(s)$. Let $s_j(0)$ be the initial positions. For $j = 1, \ldots, J$, play the game using MCTS:

$$\vec{\pi}_j(a|s_j(k)) = \text{MCTS}(s_j(k), f_\theta; M) \tag{3.39}$$

$$s_j(k+1) = \arg\max_a \vec{\pi}^*_j(a|s_j(k)) \tag{3.40}$$

for $k = 0, \ldots, K_j$. The game ends at

$$K_j = \inf\{k|s_j(k) \in \mathcal{T}_{\text{end}}\} \tag{3.41}$$

Then the "true" policy and value are

$$\vec{\pi}^*_j(k) = \vec{\pi}_j(\cdot|s_j(k)) \tag{3.42}$$

$$v^*_j(k) = \begin{cases} 1 & K_j > K_{\max} \\ -1 & \text{otherwise} \end{cases} \tag{3.43}$$

The parameters θ of the neural network are updated by stochastic gradient descent (SGD) on the loss function:

$$\begin{aligned} &\min_{\theta} \sum_{j=1}^{J} \sum_{k=0}^{K_j} L_{\text{policy}}\Big(\vec{\pi}_\theta(s_j(k)), \vec{\pi}_j^*(k)\Big) + L_{\text{value}}\Big(v_\theta(s_j(k)), v_j^*(k)\Big) \\ &\qquad L_{\text{policy}}(\vec{p}, \vec{q}) = -\vec{p} \cdot \log \vec{q} \\ &\qquad L_{\text{value}}(p, q) = (p - q)^2 \end{aligned} \tag{3.44}$$

We use a learning rate of 0.001 and the Adam optimizer [47].

3.4.4 Numerical results

A key difficulty in learning a good policy for the pursuer is that it requires a good evader. If the evader is static, then the pursuer can win with any random policy.

During training and evaluation, the game is played with the evader moving according to (3.30). Although all players move simultaneously, our MCTS models each team's actions sequentially, with the pursuers moving first. This is conservative towards the pursuers, since the evaders can counter.

We train using a single workstation with 2 Intel Xeon CPU E5-2620 v4 2.10GHz processors and a single NVidia 1080-TI GPU. For simplicity, f_P and f_E are constant, though it is straightforward to have spatially varying velocity fields. We use a gridsize of $m = 16$. We set $K_{\max} = 100$ and $M = 1000$ MCTS iterations per move. One step of training consists of playing $J = 64$ games and then training the neural network for 1 epoch based on training data for the last 10 steps of training. Self-play game data is generated in parallel,

while network training is done using the GPU with batch size 128. The total training time is 1 day.

The training environments consist of between 2 to 6 randomly oriented obstacles, each uniformly chosen from the set of ellipses, diamonds, and rectangles. We emphasize that the environments shown in the experiments are not in the training set.

We compare our trained neural network against uniform random and dirichlet noise-based policies, as well as the local policies from section 3.3. In order to draw a fair comparison, we make sure each action requires the same amount of compute time. Each MCTS-based move in the 2 player game takes 4 secs while the multiplayer game takes about 10 secs per move, on average. Since the noise-based policies require less overhead, they are able to use more MCTS iterations. The shadow strategies become very expensive as more players are added. For the 1v1 game, we use $\hat{M} = 1000$, while the 2v2 game can only run for $\hat{M} = 250$ in the same amount of time as the Neural Net. Specifically,

- Distance strategy
- Shadow strategy
- Blend strategy
- MCTS$(\cdot, f_{\text{distance}}, 1000)$ where $f_{\text{distance}}(s) = (p_{\text{distance}}, 0)$.
- MCTS$(\cdot, f_{\text{shadow}}, \hat{M})$ where $f_{\text{shadow}}(s) = (p_{\text{shadow}}, 0)$.

- MCTS$(\cdot, f_{\text{blend}}, \hat{M})$ where $f_{\text{blend}}(s) = (p_{\text{blend}}, 0)$.
- MCTS$(\cdot, f_\mu, 2000)$ where $f_\nu(s) = (\text{Uniform}, 0)$.
- MCTS$(\cdot, f_\eta, 2000)$ where $f_\eta(s) = (\text{Dir}(0.3), 0)$.
- MCTS$(\cdot, f_\theta, 1000)$ where f_θ is the trained Neural Network

Two players

As a sanity check, we show an example on a single circular obstacle with a single pursuer and single evader. As we saw from the previous section, the pursuer needs to be faster in order to have a chance at winning. We let $f_P = 2$ and $f_E = 1$. Figure 3.10 shows an example trajectory using Neural Net. The neural network model gives reasonable policies. Figure 3.11 shows an adversarial human evader playing against the Neural Net pursuer, on a map with two obstacles. The pursuer changes strategies depending on the shape of the obstacle. In particular, near the corners of the "V" shape, it maintains a safe distance rather than blindly following the evader.

In order to do a more systematic comparison, we run multiple games over the same map and report the game time statistics for each method. We fix the pursuer's position at $(1/2, 1/4)$ and vary the evader's initial location within the free space. Figure 3.12 shows the setup for the two maps considered for the statistical studies in this section. One contains a single circle in the center, as we have seen previously. The other one contain 5 circular obstacles, though the one in the center has some extra protrusions.

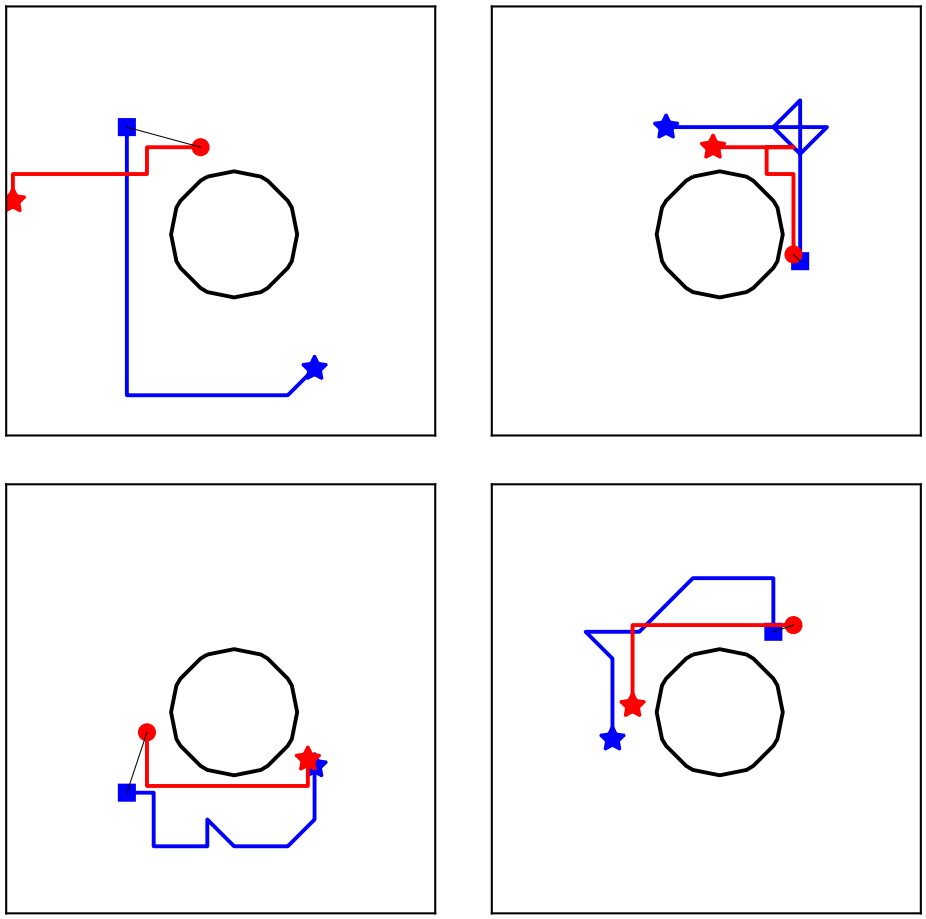

Figure 3.10: Snapshots of the trajectory for the Neural Net pursuer around a circular obstacle. The pursuer (blue) tracks the evader (red) while maintaining a safe distance. View from left to right, top to bottom. Stars indicate the initial positions, and the black line (of sight) connects the players at the end of each time interval.

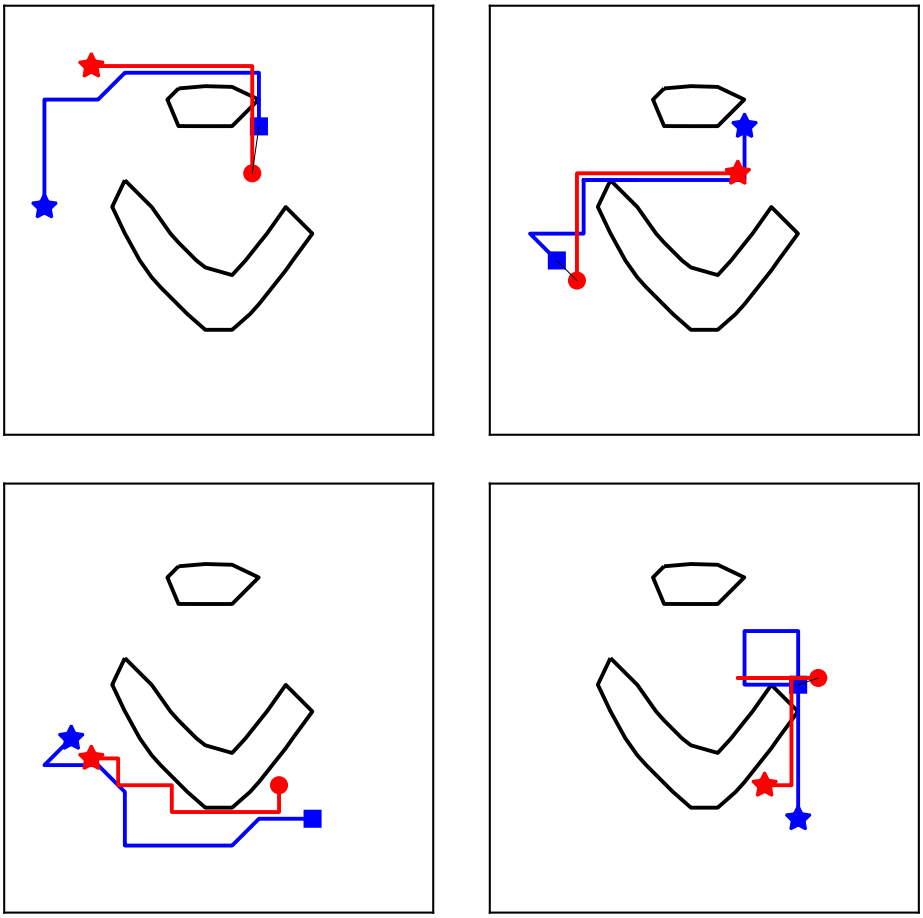

Figure 3.11: Trajectory for the Neural Net pursuer against an adversarial human evader on a map with two obstacles. The pursuer transitions between following closely, and leaving some space, depending on the shape of the obstacle.

Figure 3.13 shows an image corresponding the the length of the game for each evader position; essentially, it is a single slice of the value function for each method. Table 3.2 shows the number of games won. Shadow strategy particularly benefits from using MCTS for policy improvements, going from 16% to 67.15% win rate. Our neural network model outperforms the rest with a 70.8% win rate.

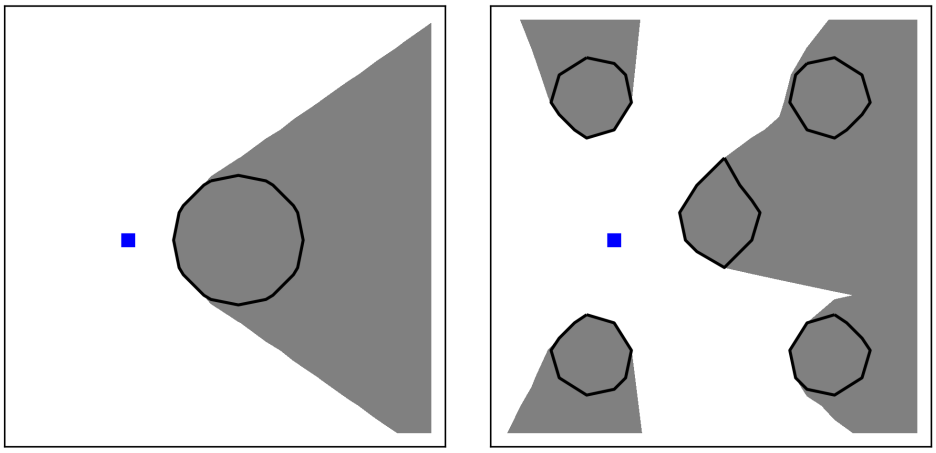

Figure 3.12: Setup for computing a slice of the value function for the circular obstacle (left) and 5 obstacle map (right). The pursuer's initial position is fixed (blue) while the evader's changes within the free space.

Multiple players

Next, we consider the multiplayer case with 2 pursuers and 2 evaders on a circular obstacle map where $f_P = 2$ and $f_E = 2$. Even on a 16×16 grid, the computation of the corresponding feedback value function would take several days. Figure 3.14 shows a sample trajectory. Surprisingly, the neural network has learned a smart strategy. Since there is only a single obstacle, it is sufficient

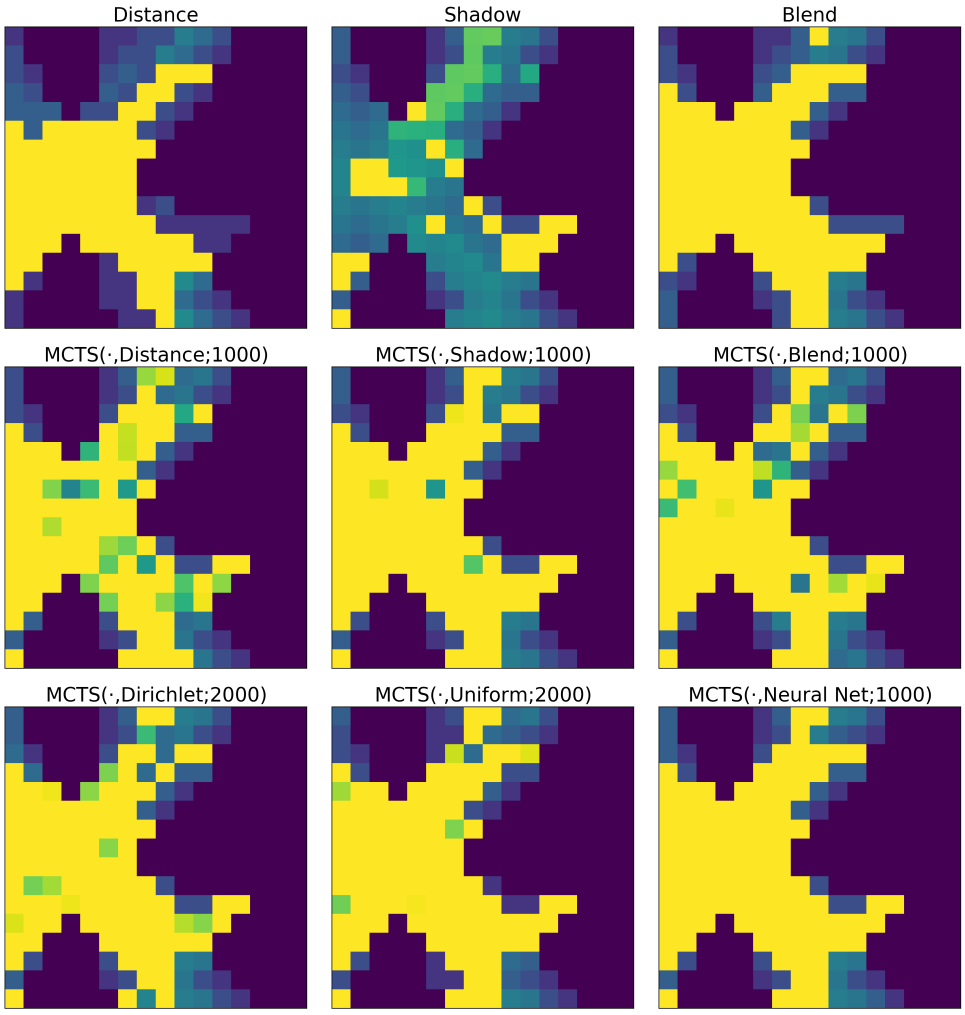

Figure 3.13: One slice of the "value" function for single pursuer, single evader game with 5 obstacles. Bright spots indicate that the pursuer won the game if that pixel was the evader's initial position.

Table 3.2: Game statistics for the 1 pursuer vs 1 evader game with 5 circular obstacles, where $f_P = 2$ and $f_E = 1$.

Method	Win % (137 games)	Average game time
Distance	52.55	53.56
Shadow	16.06	22.36
Blend	63.50	64.45
MCTS(·, Distance; 1000)	55.47	62.40
MCTS(·, Shadow; 1000)	67.15	69.45
MCTS(·, Blend; 1000)	58.39	63.27
MCTS(·, Uniform; 2000)	60.58	65.84
MCTS(·, Dirichlet; 2000)	65.69	69.02
MCTS(·, Neural Net; 1000)	70.80	71.61

for each pursuer to guard one opposing corner of the map. Although all players have the same speed, it is possible to win.

Figure 3.15 shows a slice of the value function, where 3 players' positions are fixed, and one evader's position varies. Table 3.3 shows the game statistics. Here, we see some deviation from the baseline. As the number of players increase, the number of actions increases. It is no longer sufficient to use random sampling. The neural network is learning useful strategies to help guide the Monte Carlo tree search to more significant paths. The distance and blend strategies are effective by themselves. MCTS helps improve performance for Distance. However, 250 iterations is not enough to search the action space, and actually lead to poor performance for Blend and Shadow. For this game setup, MCTS(Distance,1000) performs the best with a 73.5% win rate, followed by Blend with 65.4% and Neural Net with 59.9%. Although the trained network is not the best in this case, the results are very promising.

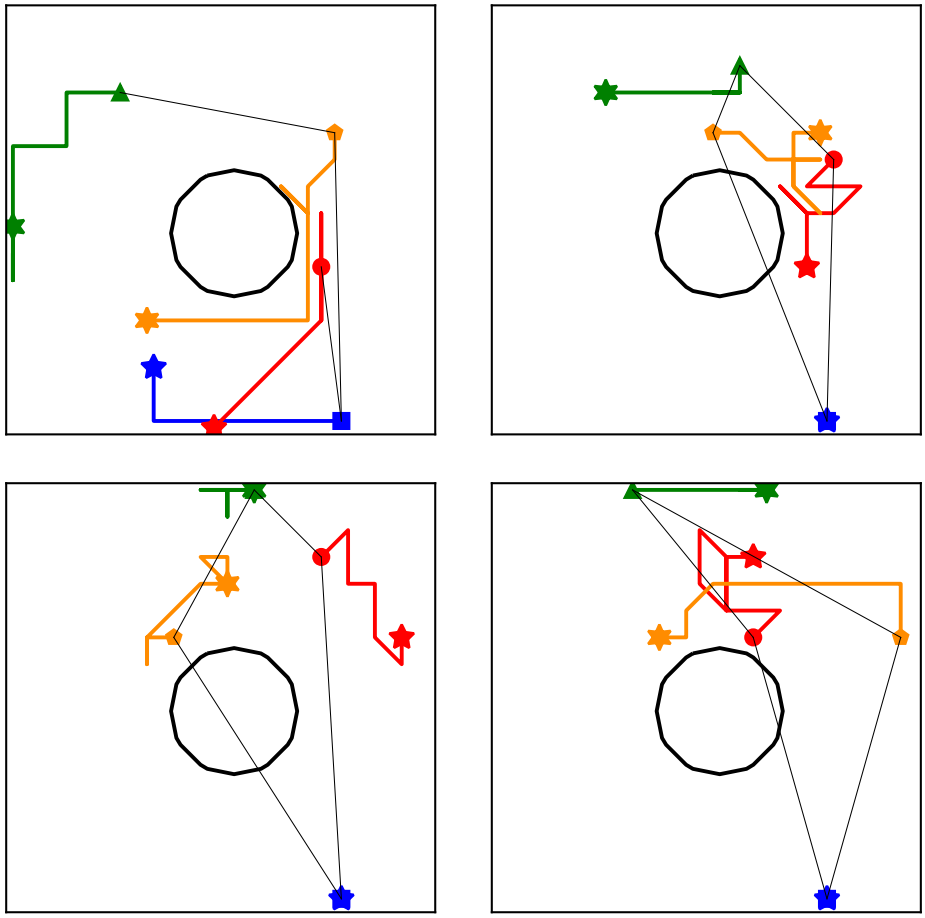

Figure 3.14: Trajectories for the multiplayer game played using NNet around a circle. Pursuers are blue and green, while evaders are red and orange. Blue has learned the tactic of remaining stationary in the corner, while green manages the opposite side. The evaders movements are sporadic because there is no chance of winning; there are no shadows in which to hide.

Table 3.3: Game statistics for the 2 pursuer vs 2 evader game with a circular obstacle.

Method	Win % (162 games)	Average game time
Distance	56.8	58.8
Shadow	46.3	50.1
Blend	65.4	67.9
MCTS(·, Distance; 1000)	73.5	76.6
MCTS(·, Shadow; 250)	40.7	44.5
MCTS(·, Blend; 250)	00.0	4.4
MCTS(·, Uniform; 2000)	00.0	5.3
MCTS(·, Dirichlet; 2000)	27.8	32.8
MCTS(·, Neural Net; 1000)	59.9	61.7

We want to emphasize that the model was trained with no prior knowledge. Given enough offline time and resources, we believe the proposed approach can scale to larger grids and learn more optimal policies than the local heuristics.

Figure 3.16 shows a comparison of the depth of search for $M = 1000$ MCTS iterations. Specifically, we report depth of each leaf node, as measured by game time. To be fair, we allow the uniform and dirichlet baselines to run for 2000 MCTS iterations to match the runtime needed for 1 move. Also, the shadow strategies are extremely costly, and can only run 250 MCTS iterations in the same amount of time. However, we show the statistics for $M = 1000$ to gain better insights. Ideally, a good search would balance breadth and depth. The neural network appears to search further than the baselines. Of course, this alone is not sufficient to draw any conclusions. For example, a naive approach could be a depth-first search.

In Figure 3.17, we show a similar chart for the single pursuer, single

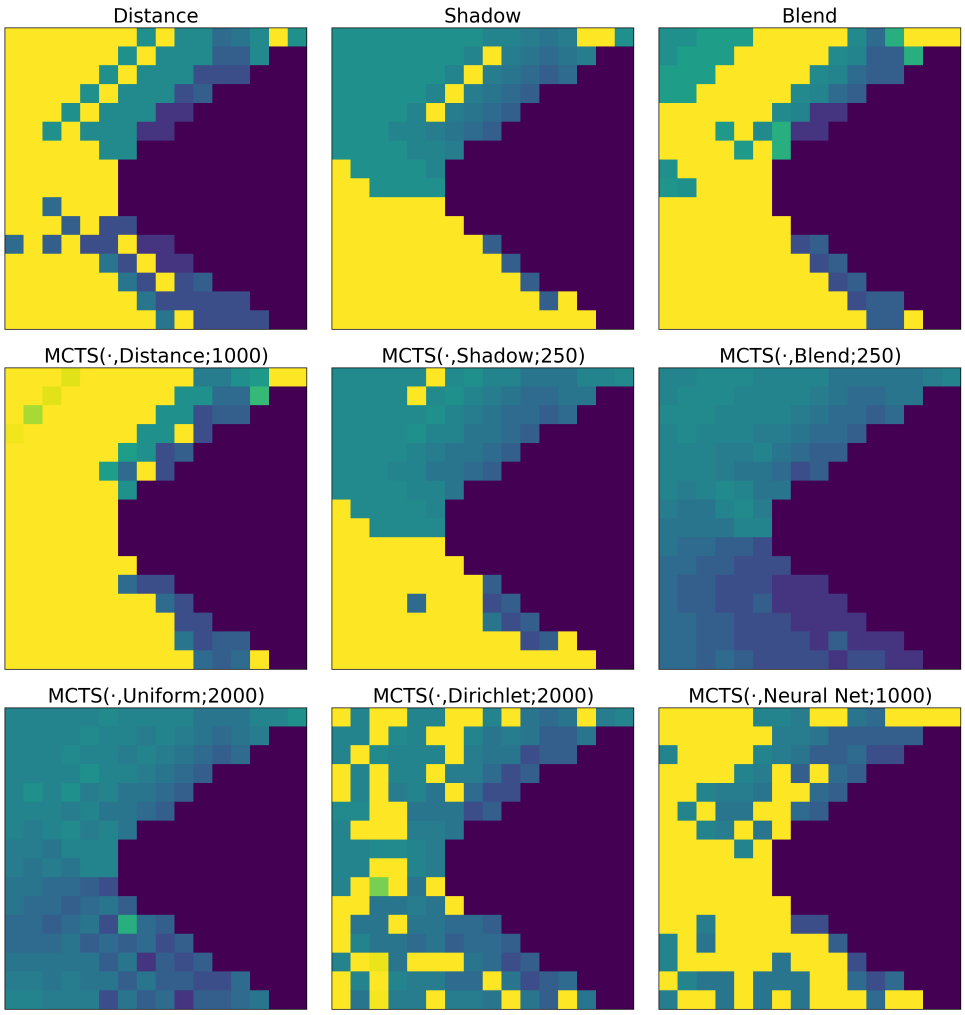

Figure 3.15: One slice of the value function for 2 pursuer, 2 evader game on the circular obstacle.

evader game with a circular obstacle. In this case, the game is relatively easy, and all evaluator functions are comparable.

3.5 Conclusion and future work

We proposed three approaches for approximating optimal controls for the surveillance-evasion game. When there are few players and the grid size is small, one may compute the value function via the Hamilton-Jacobi-Isaacs equations. The offline cost is immense, but on-line game play is very efficient. The game can be played on the continuously in time and space, since the controls can be interpolated from the value function. However, the value function must be recomputed if the game settings, such as the obstacles or player velocities, change.

When there are many players, we proposed locally optimal strategies for the pursuer and evader. There is no offline preprocessing. All computation is done on-line, though the computation does not scale well as the velocities or number of pursuers increases. The game is discrete in time and space.

Lastly, we proposed a reinforcement learning approach for the multi-player game. The offline training time can be enormous, but on-line game play is very efficient and scales linearly with the number of players. The game is played in discrete time and space, but the neural network model generalizes to maps not seen during training. Given enough computational resources, the neural network has the potential to approach the optimal controls afforded by the HJI equations, while being more efficient than the local strategies.

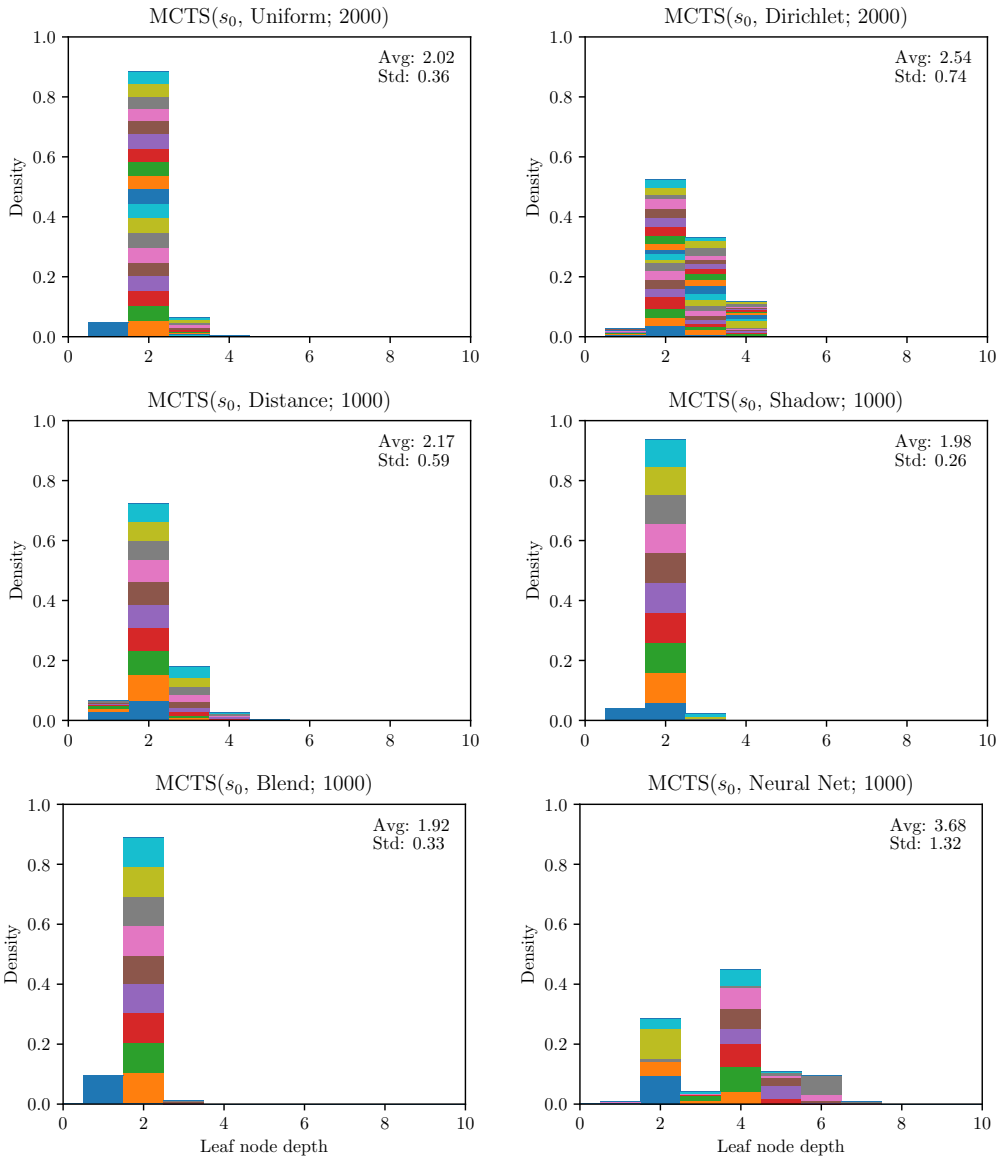

Figure 3.16: Histogram of leaf node depth for MCTS using various evaluator functions for the multiplayer game around a circular obstacle. The colors show increments of 100 iterations. The multiplayer game has a much larger action space, making tree search difficult. The neural network appears to search deeper into the tree.

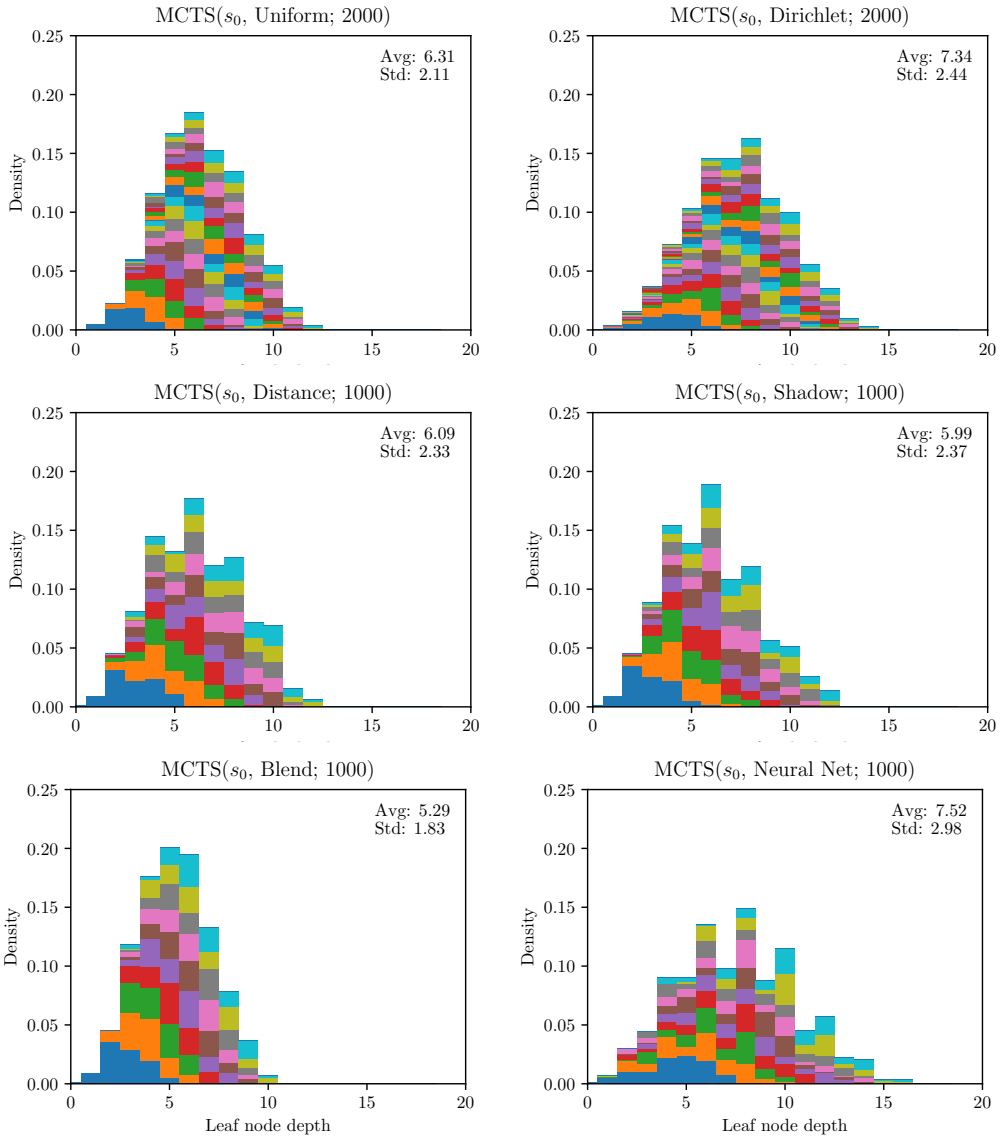

Figure 3.17: Histogram of leaf node depth for MCTS using various evaluator functions for the single pursuer vs single evader game around a circular obstacle. The colors show increments of 100 iterations. The game is relatively easy and thus all algorithms appear comparable. Note that Uniform and Dirichlet are allowed 2000 MCTS iterations, since they require less overhead to run.

There are many avenues to explore for future research. We are working on the extension of our reinforcement learning approach to 3D, which is straight-forward, but requires more computational resources. Figure 3.18 shows an example surveillance-evasion game in 3D. Along those lines, a multi-resolution scheme is imperative for scaling to higher dimensions and resolutions. One may also consider different game objectives, such as seeking out an intially hidden evader, or allowing brief moments of occlusion.

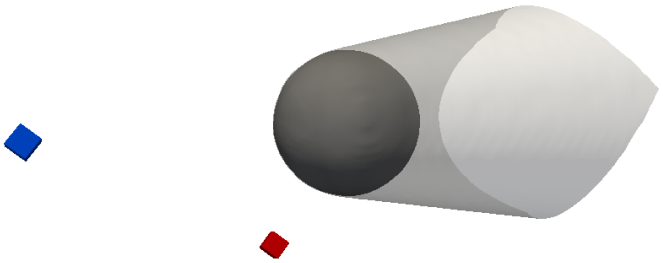

Figure 3.18: A snapshot of a 3D surveillance-evasion game around a sphere.

3.5.1 The surveillance-constrained patrol problem

In [10], we considered a surveillance-constrained patrol problem where a pursuer must optimize short-term visibility of the environment, with the constraint that it must always keep the evader within its line-of-sight. In this section, we briefly review the ideas in that paper, and mention a direction for future work which combines the ideas presented in Chapters 2 and 3.

The game is played in discrete space and time. We assume that both

players have a map of the environment; the pursuer must be faster than the evader, otherwise it may not have any flexibility to do anything other than the surveillance constraint. Formally, let $K \in \mathbb{N}$. Short-term visibility means that the pursuer's visibility of the environment at time t is only valid for K time steps, after which those portions are assumed to be occluded again. Define the short-term visibility set

$$\Omega^i_{i-K} := \bigcup_{j=i-K}^{i} \mathcal{V}(P_j) \tag{3.45}$$

As in Chapter 2, one may define the gain function

$$g^K(x; \Omega^i_{i-K}) := |\mathcal{V}_x \cup \Omega^i_{i-K}| - |\Omega^i_{i-K}|, \tag{3.46}$$

Then the problem can be stated as a constrained optimization problem:

$$\begin{aligned} &\max_{P(i)} g^K(P(i); \Omega^i_{i-K}) \\ &\text{subj. to } E(t) \in \mathcal{V}(P(t)) \text{ for } t \geq i \end{aligned} \tag{3.47}$$

The constraint is challenging since it needs to hold for all future time. In [10], we satisfy the constraint by using reactive synthesis tools to precompute the *feasible set*

$$\mathcal{F} := \{(P(0), E(0)) | E(t) \in \mathcal{V}(P(t)) \text{ for } t \geq 0\},$$

which is the set of positions from which it is possible to maintain the surveillance requirement for all time. The problem can now be reformulated as

$$\begin{aligned} &\max_{P(i)} g^K(P(i); \Omega^i_{i-K}) \\ &\text{subj. to } (P(i), E(i)) \in \mathcal{F}, \end{aligned} \tag{3.48}$$

where the optimization of the objective function can be done greedily during game-play. Figure 3.19 shows snapshots from an example surveillance-constrained patrol game.

For future work, we envision the use of the value function from section 3.2 to compute the feasible set for the *continuous* version of the game. Optimization of g^K can be done using a greedy approach powered by a convolutional neural network, as in Chapter 2. Monte Carlo tree search may also help refine strategies and generate causually relevant training data that surpasses the one-step lookahead policy afforded by the greedy algorithm.

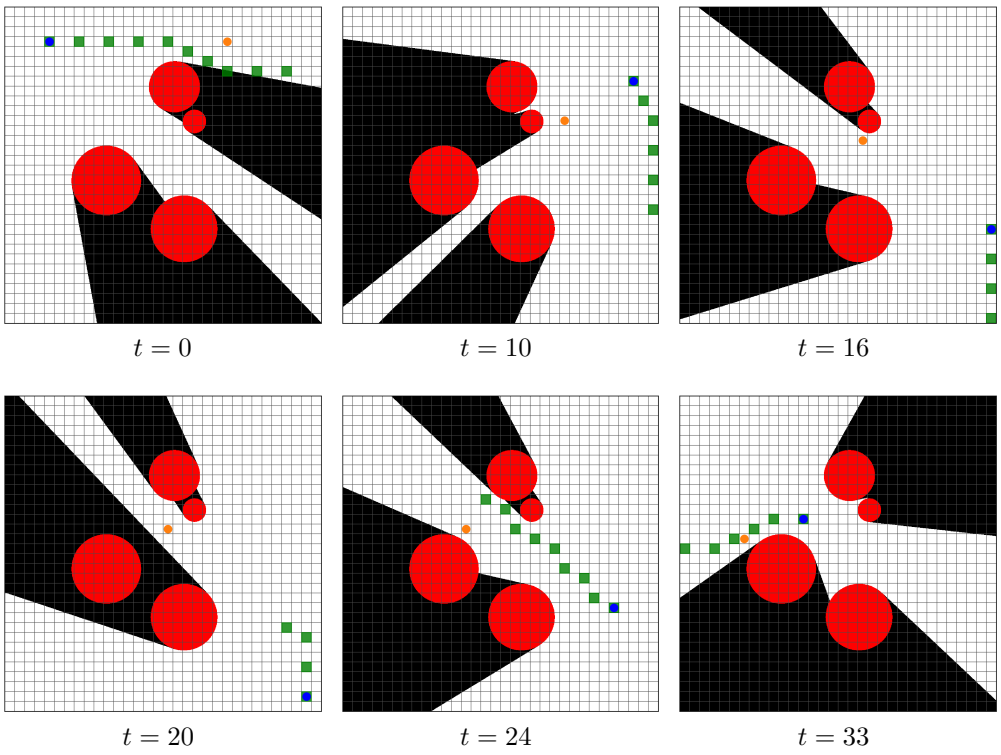

Figure 3.19: Example of the surveillance-constrained patrol game with a single pursuer (blue) and single evader (orange). The pursuer tries to optimize short-term visibility of the environment when possible, but must always maintain visibility of the evader. The green cells correspond to the pursuer's planned path to a vantage point. Obstacles are shown in red, with occlusions in black.

Chapter 4

RaySense

4.1 Introduction

In the previous chapters, we discussed problems involving line-of-sight visibility. The shadows formed by the obstacles depend intricately on the shape of the obstacles. Thus, the corresponding algorithms for exploration and surveillance-evasion are functions over the space of shapes. This chapter considers something slightly different: a way to sample and characterize shapes. Admittedly, it is not entirely relevant to the notion of line-of-sight visibility. But abstractly, one can consider this chapter as relating to the visibility of point clouds.

In particular, we propose a novel method for sampling point clouds or other geometric objects. We call our approach "RaySense" because it samples by firing randomly-chosen rays through the ambient space occupied by the object. At a few points along each ray, we sample the nearest neighbors in the object; the ray senses the structure of the object. We can then work with

This chapter contains portions of a preprint completed in collaboration with my advisor Richard Tsai, as well as Liangchen Liu and Colin Macdonald from the Department of Mathematics at the University of British Columbia. The collaboration initiated at the Institute for Pure and Applied Mathematics (IPAM).

this data—the *RaySense signature*—instead of the original object. The size of the RaySense signature can be predetermined, while the size of each point cloud may vary. Therefore, the use of RaySense signatures offers flexibility in designing algorithms for comparing data sets of different sizes.

RaySense assumes that the object, Γ, is embedded in a Euclidean space, $\mathbb{R}^d$ using some suitable representation. For example, $d = 3$ for CAD models of a physical object [18, 114], an implicit surface, or a collection of shapes, discretized and represented as black-and-white images. Γ may contain point sets from geometrical objects containing parts of different Hausdorff dimensions; e.g., solids balls inter-connected by line segments. In general, RaySense will work on data already transformed into a suitable feature space. In this chapter, we limit much of the discussion to point sets in $\mathbb{R}^3$. We assume the objects to be compared are calibrated, e.g., centered, rotated, and scaled consistently.

When the object is a point cloud, the RaySense samples are easy to find via discrete nearest neighbor searches. There are computationally efficient algorithms for performing nearest-in-Euclidean-distance neighbor searches, for example, tree-based algorithms [8], and grid-based algorithms [105]. For very high dimensions, there is also randomized nearest neighbor search algorithms [42]. We show that certain statistical information from the sampled data are dependent only on the ray distribution, not specific ray sets.

When the object is a smooth submanifold in $\mathbb{R}^d$, one can easily extract local geometrical information (such as curvature) from the nearest points of the rays. When Γ is a finite point set on a smooth manifold, curvature information

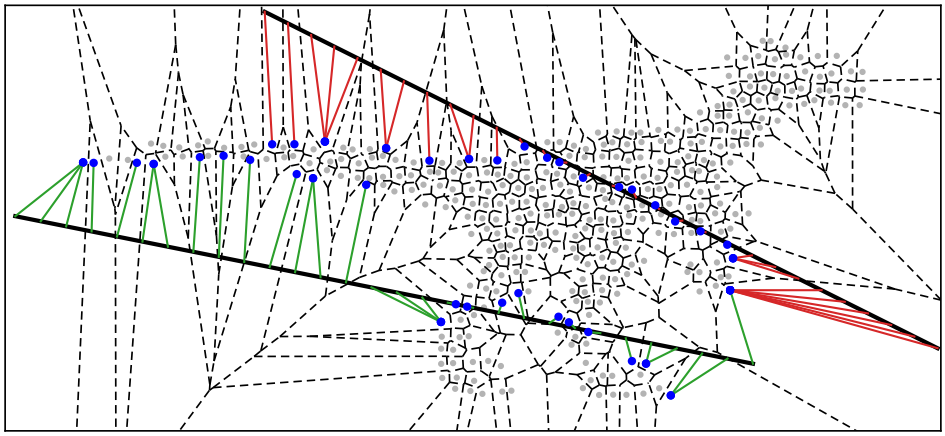

Figure 4.1: A simple 2D point set (gray). Two rays (black) sense nearest neighbors of the point set (blue). Singular points, such as the tip of the tail, have larger Voronoi cells (dashed lines) and are more likely to be sampled. Closest point pairs are shown in green and red.

can also be derived from multiple nearest neighbors of each point on the rays.

RaySense has potential applications for *registration*, *classification*, *segmentation*, and *compression* of data. In this chapter, we consider registration as a pre-processing step that is already applied to the data set, and focus on the problem of classification.

Contributions

We provide theoretical and empirical analysis on properties of our proposed sampling framework, including statistical invariance, independence to embedding dimension and repeated sampling of salient feature points. We design RayNN, a neural network with 1D convolutions tailored for RaySense. We evaluate our network on benchmark point cloud classification datasets

and show that, compared with other state-of-the-art methods, RayNN is more robust against unseen outliers and has lower complexity while achieving comparable accuracy. Intuitive explanations for RaySense's success include (a) repeated sampling of salient feature points; and (b) some locality and high-order information related to (suitably-defined notions) of curvatures.

4.1.1 Related work

In Integral Geometry, one uses the probability of intersection of affine subspaces of different dimensions with the target data manifold to deduce information about the manifold. The interaction information obtained from the "sensing" affine subspaces is binary: yes or no; i.e., $X = \{0, 1\}$. One thus has a counting problem: how frequently will rays intersect with the data manifold. From these probabilities, one may extract geometrical information about the manifold; see e.g., [48]. Nevertheless, this approach may be inefficient in practice. One may further consider integrating certain information gathered along rays. The Radon transform is a classic example where a local density is integrated along each ray.

Our idea is to add additional dimensions to record information about the data. For example, along a ray, we may store the distance to the closest point in the data. One can draw an analogy to seismic imaging, where designated points on each ray correspond to geophones that record the first arrival time of waves from known sources. One difference is that in seismic imaging, the sensor arrays typically lie on top of the domain of interest—in RaySense,

the sensors are placed on rays that penetrate the ambient space.

Abstractly speaking, RaySense is about the mapping of a set of randomly selected rays to some space X that is used to record information about the data. Correspondingly, one designs functions on X to extract information. In RaySense, we propose the use of nearest neighbor information.

Salient points tend to appear more frequently within the RaySense signature. By retaining only the most frequently repeated points, RaySense resembles keypoint detectors [51] or compression algorithms, such as autoencoders and principal component analysis (PCA). However, autoencoders are less interpretable which leads to difficulties in theoretical analysis. PCA projects data onto a lower-dimensional space, resulting in loss of important geometric information.

From the perspective of the computer vision community, RaySense can be considered as a shape descriptor, mapping from 3D point sets to a more informative feature space where point sets can be easily compared. Generally, descriptors try to capture statistics related to local and global features. See [46] for a survey.

More recently, researchers have combined shape descriptors with machine learning [27, 81, 93, 115]. Others designed neural networks to learn the shape descriptors directly from point clouds [1, 49, 64, 65, 77, 78, 91, 112, 113, 120]. PointNet [77] pioneered deep learning on point clouds by applying independent operations on each point and aggregating features via a symmet-

ric function. Based on that, other architectures [78, 88] exploit neighboring information to extract local descriptors. SO-Net [64] uses self-organizing maps to hierarchically group nodes while applying fully-connected layers for feature extraction. PCNN [1] defines an extension and pulling operator similar to the closest point method [82, 67] to facilitate the implementation of regular convolution, while DGCNN [113] and PointCNN [65] generalize the convolution operator to point sets.

Much of the research above is on 3D point clouds. RaySense applies more generally to data in arbitrary dimensions. Rather than using machine learning on the point set, we use the RaySense signature as input. We show this is more efficient for classification in § 4.4.

The ray-casting and ray-tracing communities [34, 53, 76] use kd-tree–based algorithms for very efficient computation of nearest-neighbor queries, curvature, and other quantities, for very large sets of rays. These techniques would be useful for improving the efficiency of RaySense implementations.

4.2 Methods

We assume all points are properly calibrated by a common preprocessing step.* For simplicity, we normalize each point set to be in the unit ℓ^2 ball, with center of mass at the origin.

*One could use learning based on RaySense to train such a preprocessor for registration; we do not explore this idea further in this work.

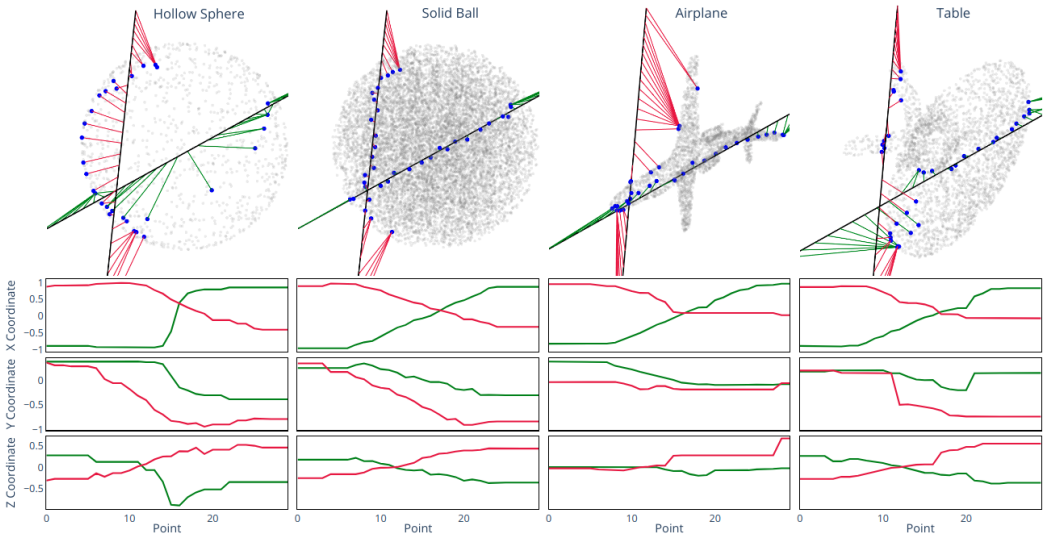

Figure 4.2: RaySense signatures using 30 sample points per ray. Row 1: visualization of two rays (black) through points sampled from various objects (gray). Closest point pairs are shown in green and red. Rows 2–4: the x, y, and z coordinates of the closest points to the ray.

We draw rays from a distribution $\mathcal{P}$. Each ray is a line segment in $\mathbb{R}^d$. We consider k uniformly spaced points along each ray, with spacing δr. With $r_{i,j}$ denoting the j-th point on the i-th ray, we define the RaySense signature tensor $S(\Gamma)$, with entries $[S(\Gamma)]_{i,j} := P_\Gamma r_{i,j}$, where $P_\Gamma r_{i,j}$ is the nearest point in Γ to $r_{i,j}$. In cases of nonuniqueness, we choose arbitrarily. Later in § 4.2 we generalize the entries of the RaySense signature to live in a "feature space" X by including additional components.

Generating random rays

We present two ways to generate random rays. There is no *right* way to generate rays, although it is conceivable that one may find optimal ray

distributions for specific applications.

Method R1

One simple approach is generating ray segments of fixed-length L, whose direction $\vec{v}$ is uniformly sampled from the unit sphere. We add a shift $\vec{a}$ sampled uniformly from $[-\frac{1}{2}, \frac{1}{2}]^d$ to avoid a bias for the origin. The k sample points are distributed evenly along the ray:

$$\vec{r_i} = \vec{a} + L\left(\frac{i}{k-1} - \frac{1}{2}\right)\vec{v}, \qquad i = 0, \ldots, k-1$$

The spacing between adjacent points on each ray is denoted by δr, which is $L/(k-1)$. We use $L = 2$.

Method R2

Another natural way to generate random rays is by random endpoints selection: choose two random points $\vec{p}$, $\vec{q}$ on a sphere and connect them to form a ray. Then we evenly sample k points between $\vec{p}$, $\vec{q}$ on the ray. To avoid overly short rays where information would be redundant, we use a minimum ray-length threshold τ to discard rays. Note that the spacing of points on each ray varies, depending on the length of the ray segment:

$$\vec{r_i} = \vec{p} + \frac{i}{k-1}(\vec{q} - \vec{p}), \qquad i = 0, \ldots, k-1.$$

In this paper, we use Method R1; a fixed δr helps maintain spatial consistency along the rays, which increases RayNN's classification accuracy in § 4.4.

What is included in the signature?

Let $f : \Gamma \times \mathbb{R}^d \mapsto X$ map points in Γ and on a ray into some "feature space" X, and assume that X is embedded in $\mathbb{R}^c$. Building on § 4.1, we generalize the RaySense signature tensor to have entries

$$[S_{m,\delta r}(\Gamma; f, \mathcal{P})]_{i,j} := f(P_\Gamma r_{i,j}, r_{i,j}).$$

We will continue to denote this RaySense signature as simply "$S(\Gamma)$".

In this work, we propose that the RaySense signature includes the coordinates of the closest point to each ray sample point and the vector to the closest point:

$$f(P_\Gamma r_{i,j}, r_{i,j}) = [P_\Gamma r_{i,j}, P_\Gamma r_{i,j} - r_{i,j}].$$

In addition, it can include the distance to the closest point $\|P_\Gamma r_{i,j} - r_{i,j}\|$. The signature can also be extended to include these features from κ nearest-neighbors. We will see that incorporating additional neighbors into the signature increases robustness to outliers.

Snapshots of RaySense features

RaySense signatures For discrete point sets, the likelihood that a ray senses a particular point is related to the volume of its Voronoi cell; this is similar to a Monte-Carlo approximation to the volume, as we discuss further in § 4.3. Fig. 4.1 shows that salient points are more likely to be sensed by a ray due to their larger Voronoi cells. In Fig. 4.2, we show examples of two rays sensing various 3D point clouds, along with the corresponding features of the signature

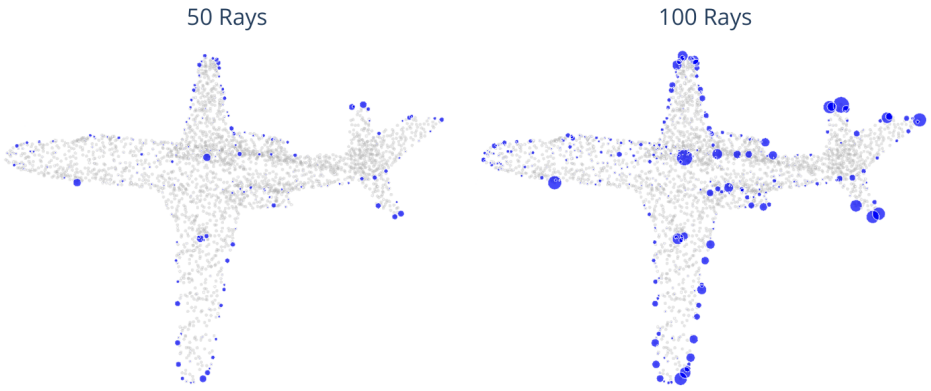

Figure 4.3: RaySense is more likely to sample salient features in the point cloud. Larger points are repeated more often. We can control the number of points by increasing the number of rays. Each ray contains 30 sample points.

tensor. Fig. 4.3 shows how often each point is "sensed" by different rays. Points that are sampled by multiple rays are larger.

Curvature information

If the object is a smooth manifold in $\mathbb{R}^d$ (e.g., the sphere in Fig. 4.2) then each ray induces a parameterized curve $\gamma(t) \in \mathbb{R}^d$. The curvature of $\gamma(t)$ can be approximated by finite differences of consecutive values along the ray. For example, a typical experiment with $\delta r = 0.05$ when Γ is a unit sphere gave values such as 0.9991, 0.9982 and 0.9974, compared to the exact value of 1.

Thus even with only a few rays, we obtain local samples of higher-order geometric information. Note the calculations in this example can be performed as combinations of 1×3 convolutions along the ray; thus we can expect RayNN in § 4.4 to have access to curvature information.

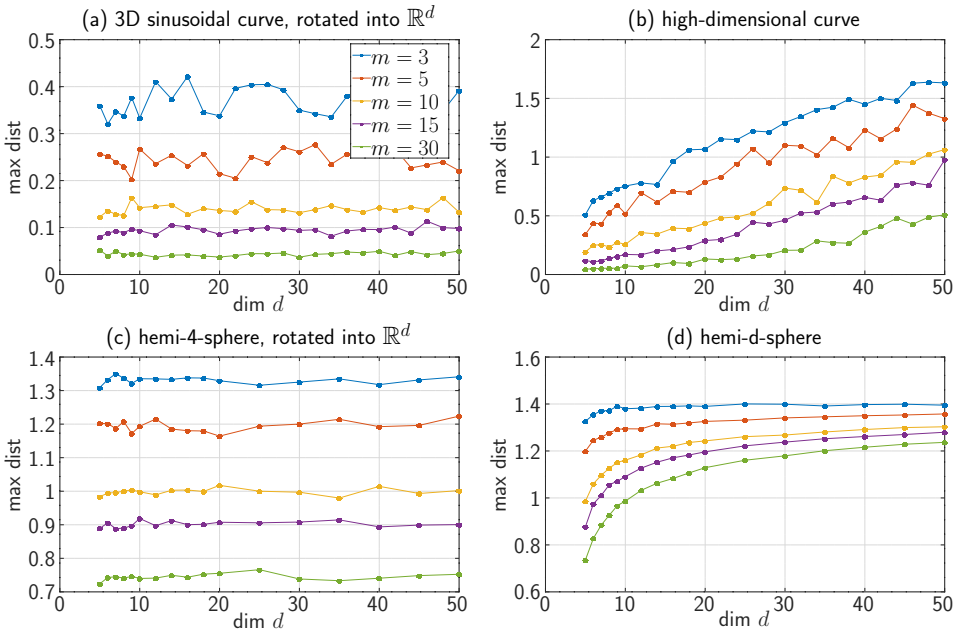

Figure 4.4: Coverage of point clouds in various dimensions by RaySense using m rays with 32 samples per ray. Top: 5000 points sampled from curves. Bottom: 25000 points sampled from hemispheres. Low-dimensional examples embedded by random rotations into $\mathbb{R}^d$. Noise of size 10^{-4} added and results averaged over 40 realizations.

"Coverage" of an object by RaySense

Our next experiments look at how well a set of rays "cover" data sets in higher dimensional Euclidean spaces. One way to measure this is to compute the maximum distance from every point in an object Γ to points that contribute to the signature $S(\Gamma)$. The smaller this value, the closer we are to sampling the entire object. We experiment with different point clouds of various dimensions in Fig. 4.4. Notably, the coverage does not strongly depend on m when the object is inherently lower dimension and merely "rotated" into the higher dimensional space (first column of Fig. 4.4). If the object is more complicated, we may need more rays to attain the same coverage as the dimension increases (Fig. 4.4 top-right). Nonetheless, we often obtain coverage that is roughly dimension-independent (Fig. 4.4 bottom-right).

Data in higher dimensions

We consider the MNIST dataset [60], treating each image as a point in $d = 784$ dimensions. Here Γ is point set consisting of all images of the same digit. As with the airplane example in Fig. 4.3, RaySense tends to sample salient points in the data. Fig. 4.5 shows the average digits over the whole dataset, versus the average of those sampled by RaySense. In the context of MNIST, salient points are digits that are drawn using less typical strokes (according to the data). These are the data points that may be harder to classify, since they appear less frequently in the data. RaySense may be used to determine the most *useful* data points to label, as in active learning [86].

Figure 4.5: Each digit averaged over the entire data set (top) versus those sampled by RaySense (bottom).

How to use $S(\Gamma)$ for classification?

A natural idea is to choose a suitable metric to define the distances between the RaySense signature tensors. Then a test data set is labeled "A" if it is closest (by the metric) to the point sets that are also labeled as "A". We offer several choices of metric.

The Frobenius norm of the signature tensor is suitable if the signatures contain the distance and the closest point coordinates. For data sampled from smooth geometries, this information along each ray are piecewise continuous. So ℓ^2-norm based comparison seems adequate.

Wasserstein distances are more appropriate for comparison of histograms of the RaySense data. The normalized histograms can be regarded as probability distributions. In particular, notice (Fig. 4.6) that RaySense histograms tend to have "spikes" that correspond to the salient points in the data set; ℓ^2 distances are not adequate for comparing distributions with such features.

Here we briefly describe the Wasserstein-1 distance, or Earth mover's distance, that we used in this paper. Let (X, μ) and $(\tilde{X}, \nu)$ be two probability

spaces and F and G be the cumulative distribution functions of μ and ν, respectively. The Wasserstein-1 distance is defined as

$$W_1(\mu, \nu) := \int_{\mathbb{R}} |F(t) - G(t)| \, \mathrm{d}t.$$

Neural network classifier

One can consider using a properly designed and trained *neural network*. In § 4.4, we present a neural network model, RayNN, for comparing point clouds in three dimensions.

4.3 Statistical invariances

If we collect the histogram of the points sampled from a set of randomly selected rays, we can show that the histogram has a well-defined limit as the number of rays tends to infinity. Let $U \in \mathbb{R}^d$ be a solid dimension-d ball, and $\Gamma \subsetneq U$ is a finite point set containing N distinct points. We draw random rays in $\mathbb{R}^d$ from a distribution $\mathcal{P}$, e.g., by Method R1.

Let V_j denote the Voronoi cell for the j-th point, x_j in Γ, as seen in Fig. 4.1. Let $L_j(\omega)$ denote the length of a ray, ω, that lies in $V_j \cap U$. If ω does not intersect V_j, $L_j(\omega) := 0$. Thus, L_j is a random variable, and we denote it's expectation by $\mathbb{E}[L_j] := \int L_j(\omega) \, \mathrm{d}P(\omega)$.

A hybrid Monte-Carlo approach can approximate $\mathbb{E}[L_j]$. Draw m rays from the distribution. On each ray, collect the closest points in Γ from equidistant points that lie within U. Let δr denote the spacing between two adjacent

points. Enumerate this set of points by r_i with $i \in \mathbb{Z}$. The closest point of r_i is x_j if $r_i \in V_j$. (If r_i lies on the boundary of different cells, we pick one randomly.) Let

$$H_j(S_{m,\delta r}(\Gamma)) := \frac{1}{m} \sum_{\ell=1}^{m} \sum_{r_i \in V_j} \delta r.$$

Here, $S_{m,\delta r}(\Gamma)$ denotes the signature tensor. This H_j is precisely the number of times x_j is sampled by the RaySense approach, normalized by $\delta r/m$. Therefore, we arrive at the following Theorem:

Theorem 4.3.1. *Convergence of RaySensed data histograms:*

$$\lim_{\substack{m\to\infty \\ \delta r\to 0}} H_j(S_{m,\delta r}(\Gamma)) = \mathbb{E}[L_j].$$

Monte-Carlo approximations of integrals converge with a rate independent of the dimension. Consequently, for sufficiently many randomly selected rays, the histogram is essentially independent of the specific rays that are used.

Similar arguments show that the sampling of any function of the data set will be independent of the actual ray set, since the histograms are identical in the limit. More precisely, suppose $g : x \in \Gamma \mapsto \mathbb{R}$ is some function, then

$$\lim_{\substack{m\to\infty \\ \delta r\to 0}} \frac{1}{m} \sum_{\ell=1}^{m} \sum_{r_i \in V_j} g(x_j)\delta r = \mathbb{E}[g(x_j)L_j].$$

Fig. 4.6 shows the histograms of the coordinates of the points from the RaySense signature of Γ.

Since the Voronoi cell depends smoothly on Γ, $\mathbb{E}[L_j]$ (or $\mathbb{E}[g(x_j)L_j]$ for continuous g) will also depend smoothly on Γ. This means that it is stable

against perturbation to the coordinates of the points in Γ. However, the effect of introducing new members to Γ, such as outliers, will be non-negligible. One possible way to overcome this is to use multiple nearest neighbors for points on the rays. Such information will be different for the outliers. The other possibility, as we shall demonstrate later in this paper, is to train a suitable neural network that is less sensitive to outlier contamination.

Comparison of histograms

We experiment by comparing Γ drawn from 16 384 objects of 16 categories from the ShapeNet dataset [18]. Let l^i be the label for object Γ_i. We compute the histogram h^i_x, h^i_y, h^i_z of the x, y, z coordinates, respectively, for points sampled by 50 rays with $k = 10$ samples per ray. We compare the histograms against those corresponding to other objects in the dataset, using

$$D_{i,j} = d(h^i_x, h^j_x) + d(h^i_y, h^j_y) + d(h^i_z, h^j_z),$$

where $d(\cdot, \cdot)$ is either the ℓ_2 or Wasserstein-1 distance. We sum D according to the respective labels

$$M_{a,b} \propto \sum_{i:l^i=a} \sum_{j:l^j=b} D_{i,j}, \qquad a, b = 1, \ldots, 16,$$

and normalize by the number of occurrences for each a, b pair. Fig. 4.7 shows the matrix of pairwise distances M between the 16 object categories.

Ideally, intra-object distances would be small, while inter-object distances would be large. As expected, Wasserstein-1 is a better metric for

comparing histograms. Still, not all objects are correctly classified. When comparing histograms in not sufficient, one may consider using higher-order statistics or neural networks to learn more complex mappings between the data and label.

4.4 Neural network for classification

We use the RaySense signature to classify objects from the ModelNet dataset [114], using a neural network, which we call RayNN.

We use a postfix notation to indicate more precisely what is included in the RaySense signature, see § 4.2. We use f with different number of neighbors, denoted by RayNN-X, where X is related to the input features. For our implementation, while we might use different numbers of nearest neighbors, we always include the closest point coordinates and the vector to closest points in our feature space ($c = 6$ fixed). We denote our models by RayNN-cp$n$ where n denotes the number of nearest neighbors.

4.4.1 Implementation details

Architecture

RayNN takes the $m \times k \times c$ RaySense signature $S(\Gamma)$ as input, and outputs a K-vector of probabilities, where K is the number of object classes. The first few layers of the network are blocks of 1D convolution followed by max-pooling to encode the signature into a single vector per ray. Convolution and max-pooling are applied along the ray. After this downsizing, we

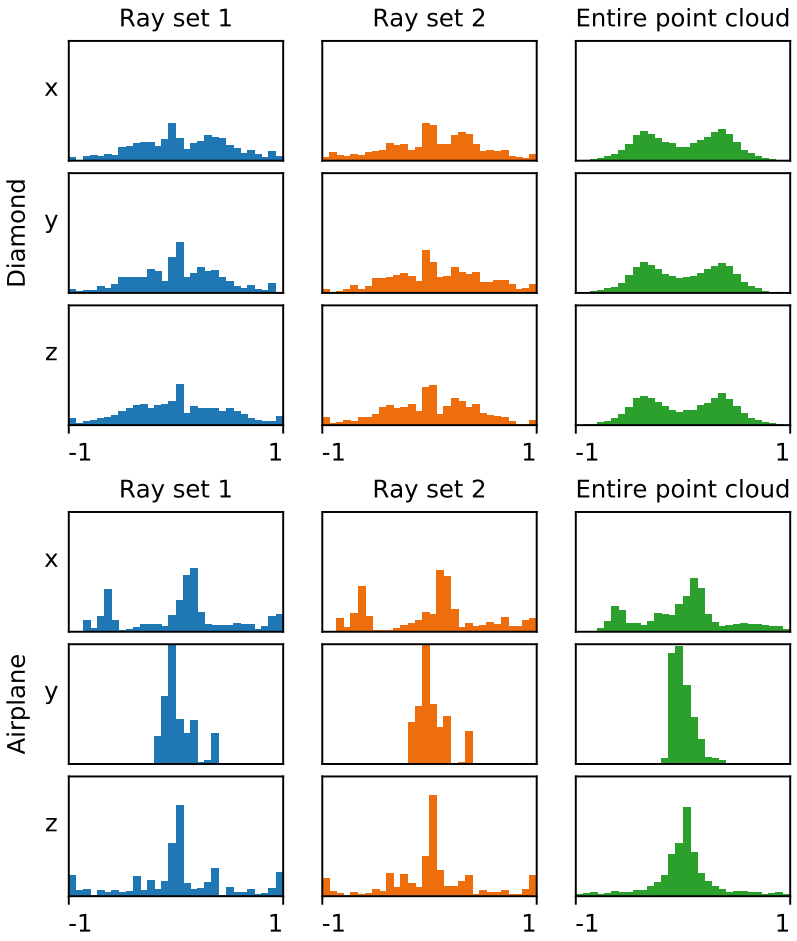

Figure 4.6: Histogram of coordinates from two point sets. Columns 1 and 2 correspond to 2 different sets of rays, each containing 50 rays and 50 samples per ray. These histograms are similar for the same object and different for different objects. Column 3 corresponds to the entire point cloud; these differ from the RaySense histograms.

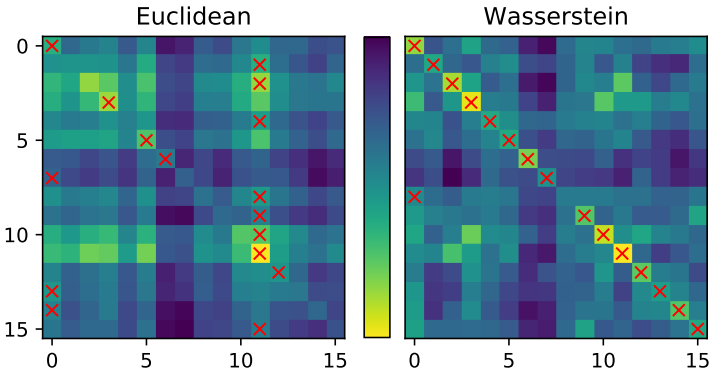

Figure 4.7: Comparison of histograms of the x, y, z coordinates of points sampled by RaySense, using ℓ^2 and Wasserstein distance W_1. Rows and columns correspond to object labels. Red $\times$ indicate location of the argmin along each row.

implement a max operation across rays. Fig. 4.8 includes some details. The output of the max pooling layer is fed into fully-connected layers with output sizes 256, 64, and K to produce the desired vector of probabilities $\vec{\mathbf{p}}_i \in \mathbb{R}^K$. Batchnorm [40] along with ReLU [70] are used for every fully-connected and convolution layer.

Note that our network uses convolution along rays to capture local information while the fully-connected layers aggregate global information. Between the two, the max operation across rays ensures invariance to the ordering of the rays. It also allows for an arbitrary number of rays to be used during inference. These invariance properties are similar to PointNet's permutation invariance [77].

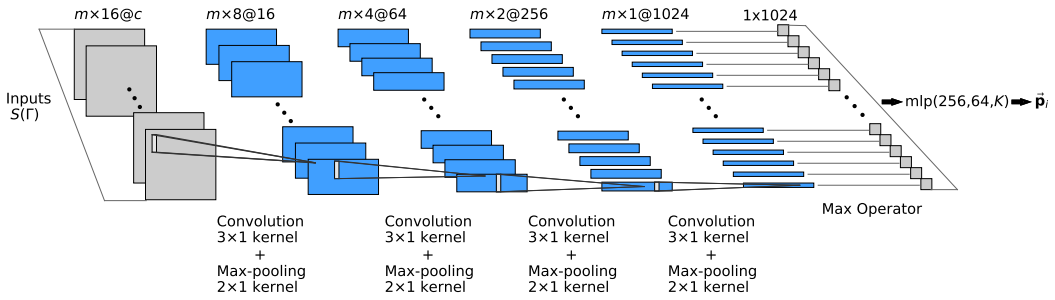

Figure 4.8: The RayNN architecture for m rays and k samples per ray. The input is c feature matrices from $S(\Gamma)$. With $k = 16$, each matrix is downsized to an m-vector by 4 layers of 1-D convolution and max-pooling. The max operator is then applied to each of the 1024 m-vectors. The length-1024 feature vector is fed into a multi-layer perceptron (mlp) which outputs a vector of probabilities, one for each of the K classes in the classification task. Note the number of intermediate layers (blue) can be increased based on k and c.

Data

We apply RayNN on the standard ModelNet10 and ModelNet40 benchmarks [114] for 3D object classification. ModelNet40 consists of 12 311 orientation-aligned [84] meshed 3D CAD models, divided into 9843 training and 2468 test objects. ModelNet10 contains 3991 training and 908 test objects. Following the experiment setup in [77], we sample $N = 1024$ points from each of these models and rescale them to be bounded by the unit sphere to form point sets.[†] Our results do not appear to be sensitive to N.

[†]RaySense does not require point clouds for inputs: we could apply RaySense directly to surface meshes, implicit surfaces, or even—given an fast nearest neighbor calculator—the CAD models directly.

Training

During training, we use dropout with ratio 0.5 on the penultimate layer. We also augment our training dataset on-the-fly by adding $\mathcal{N}(0, 0.0004)$ noise to the coordinates. We use Adam optimizer [47] with momentum 0.9 and batch size 16. The learning rate starts at 0.002 and is halved every 100 epochs.

Inference

Our algorithm uses random rays, so it is natural to consider strategies to reduce the variance in the prediction. We consider one simple approach during inference by making an ensemble of predictions from λ different ray sets. The ensemble prediction is based on the average over the λ different probability vectors $\vec{\mathbf{p}}_i \in \mathbb{R}^K$, i.e.,

$$\text{Prediction}(\lambda) = \frac{1}{\lambda} \sum_{i=1}^{\lambda} \vec{\mathbf{p}}_i.$$

The assigned label then corresponds to the entry with the largest probability. We denote the number of rays used during training by m, while the number of rays used for inference is $\hat{m}$. Unless otherwise specified, we use $\lambda = 8$, $m = 32$ rays, and $\hat{m} = m$.

4.5 Numerical results

We compare with some state-of-the-art methods for 3D point cloud classification tasks. In addition to the results reported by [77], we also compare against PointNet.pytorch, a PyTorch reimplementation [28] of PointNet. In

Table 4.1: ModelNet classification results. Here we report our best accuracy results over all experiments. For reference, the test scores for RayNN-cp5 ($m = 32$) has mean around 90.31% and standard deviation around 0.25% over 600 tests.

	ModelNet10	ModelNet40
PointNet [77]	–	89.2
PointNet++ [78]	–	90.7
ECC [91]	90.8	87.4
kd-net [49]	93.3	90.6
PointCNN [65]	–	92.5
PCNN [1]	94.9	92.3
DGCNN [113]	–	**92.9**
RayNN-cp1 ($m = 16$)	94.05	90.84
RayNN-cp5 ($m = 32$)	**95.04**	90.96
RayNN-cp5 ($m = 64, N = 4096$)	–	91.86

all our experiments, we report overall accuracy. Table 4.1 shows RayNN is competitive while enjoying lower complexity. To investigate the robustness of our network, we perform several more experiments.

Robustness to sample size

We repeat the experiments in [77, 113] whereby, after training, data is randomly removed prior to testing on the remaining points. The results in Fig. 4.9 show that RayNN performs very well despite missing significant data.

Using fewer rays

We experiment with training using a full set of $m = 32$ rays but test using smaller number $\hat{m}$ of rays. Table 4.2 shows that RayNN can achieve a

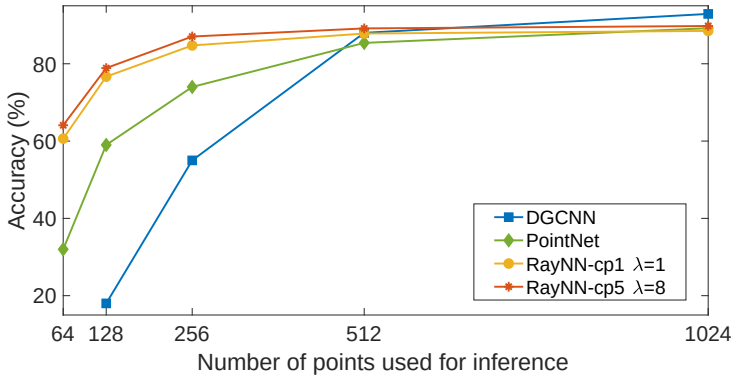

Figure 4.9: Testing DGCNN [113], PointNet [77] and RayNN on ModelNet40 with missing data.

reasonable score even if only $\hat{m} = 4$ rays are used for inference.

Robustness to outliers

This experiment simulates situations where noise (outliers) severely perturbs the original data during testing. We provide two possible solutions to tackle outliers: (a) RayNN-cp1-3rd records features from only the third nearest neighbor, and (b) RayNN-cp5 records features from all 5 nearest neighbors. We compare performance of RayNN with PointNet.pytorch in Table 4.3. The comparison reveals RaySense's capability in handling unexpected outliers, especially when additional nearest neighbors are used. RayNN-cp5 is the most robust against outliers; RayNN-cp1-3rd is more robust than RayNN-cp1 despite both having similar computational costs. Note the experiment here is different from that in [77] where the outliers are fixed and included in the training set.

Table 4.2: Accuracy when testing with a reduced ray set. RayNN-cp1 was trained using $m = 32$ rays. Results averaged over 5 runs.

ModelNet40				
$\hat{m}$	32	16	8	4
$\lambda = 1$	88.50%	86.13%	74.64%	43.28%
$\lambda = 8$	89.77%	88.94%	82.97%	55.24%

Table 4.3: Outliers sampled uniformly from the unit sphere are introduced during testing. The networks are trained without outliers. Results averaged over 5 runs.

ModelNet10	no outliers	5 outliers	10 outliers
RayNN-cp1	93.26%	79.76%	53.94%
RayNN-cp5	**93.85%**	**92.66%**	**90.90%**
PointNet.pytorch	91.08%	48.57%	25.55%
ModelNet40			
RayNN-cp1	89.77 %	54.66%	20.95%
RayNN-cp1-3rd	88.66 %	83.77%	66.36%
RayNN-cp5	**90.38%**	**88.49%**	**78.06%**
PointNet.pytorch	87.15%	34.05%	17.48%

4.5.1 Complexity analysis

Table 4.4 shows that RayNN has an advantage in model size and feed-forward time even against the simple and efficient PointNet. In both training and testing, there is some overhead in data preprocessing to build a kd-tree, generate rays, and perform the nearest-neighbour queries to form the Ray-Sense signature. For point clouds of around $N = 1024$, these costs are not too onerous in practice.

The convolution layers have $48c + 840\,016$ parameters, where c is the dimension of input feature space. The fully-connected layers have $64K + 278\,528$

parameters, where K is the number of output classes. In total, our network has $1.1 \times 10^6 + 48c + 64K \approx 1.1$M parameters. In comparison, PointNet [77] contains 3.5M parameters.

4.6 Conclusion

We present a new approach for 3D point cloud classification. Underlying our approach is a data sampling technique, RaySense, based on projecting random rays onto a data set. The projection is done by finding nearest neighbors in the data for each point along the ray. These nearest neighbors—augmented with additional features—form the "RaySense signature", which can be used for data processing.

RaySense samples salient features of the data set, such as corners or edges, with higher probability. From the RaySense signature, local information can be recovered. However, nearest-neighbor information is sensitive to outliers; using multiple nearest neighbors enhances RaySense's capability to capture persistent features in the data set, thereby improving robustness.

We have shown theoretically that the statistics of a sampled point cloud depends only on the distribution of the rays, but not on a particular ray set. The complexity of RaySense involves the numerical resolution of a single ray and Monte-Carlo sampling of the ray-distribution. The product of these is suggestive of independence of the dimension of the data embedding space. Our experience with three-dimensional data indicates not too many rays are needed in practice.

Table 4.4: Top: storage and timings for RayNN-cp1 and PointNet.pytorch on ModelNet40 using one Nvidia 1080-Ti GPU and batch size 32. The preprocessing and forward time are both measured per batch. Bottom: data from [113] is included only for reference; no proper basis for direct comparison.

	Model size	Forward time	Preprocessing	per epoch
PointNet.pytorch	14 MB	12 ms	**3.6** ms	**14** s
RayNN-cp1	**4.5** MB	**2** ms	7.5 ms	22 s
PointNet [77]	40 MB	16.6 ms	-	-
PCNN [1]	94 MB	117 ms	-	-
DGCNN [113]	21 MB	27.2 ms	-	-

We presented a neural network classifier called "RayNN". RayNN takes the RaySense signatures as input for classification of point clouds in three dimensions. We compared its performance to several other prominent models. RayNN is lightweight, flexible, efficient, and different from conventional models; for the same data set, one can test multiple times with different ray sets.

To the best of our knowledge, RaySense is a new idea, so there are many avenues of possible study. On the theoretical side, one could study RaySense's invariant properties for more general geometric objects beyond point clouds, and its connections to topological structure. There are many practical applications to explore such as the simultaneous registration, classification and segmentation of point clouds. Finally, we expect to find applications to high-dimensional data sets. For example, RaySense could be used as intermediate step for the semi-guided identification of appropriate feature spaces.

Bibliography

[1] Matan Atzmon, Haggai Maron, and Yaron Lipman. Point convolutional neural networks by extension operators. *arXiv preprint arXiv:1803.10091*, 2018.

[2] Shi Bai, Fanfei Chen, and Brendan Englot. Toward autonomous mapping and exploration for mobile robots through deep supervised learning. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2379–2384. IEEE, 2017.

[3] Tim Bailey and Hugh Durrant-Whyte. Simultaneous localization and mapping (slam): Part ii. *IEEE Robotics & Automation Magazine*, 13(3):108–117, 2006.

[4] Martino Bardi and Italo Capuzzo-Dolcetta. *Optimal control and viscosity solutions of Hamilton-Jacobi-Bellman equations*. Springer Science & Business Media, 2008.

[5] Martino Bardi, Maurizio Falcone, and Pierpaolo Soravia. Numerical methods for pursuit-evasion games via viscosity solutions. In *Stochastic and differential games*, pages 105–175. Springer, 1999.

[6] Andrew R Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information theory*,

39(3):930–945, 1993.

[7] Andrew R Barron. Approximation and estimation bounds for artificial neural networks. *Machine learning*, 14(1):115–133, 1994.

[8] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 1975.

[9] Suda Bharadwaj, Rayna Dimitrova, and Ufuk Topcu. Synthesis of surveillance strategies via belief abstraction. In *2018 IEEE Conference on Decision and Control (CDC)*, pages 4159–4166. IEEE, 2018.

[10] Suda Bharadwaj, Louis Ly, Bo Wu, Yen-Hsi Richard Tsai, and Ufuk Topcu. Strategy synthesis for surveillance-evasion games with learning-enabled visibility optimization. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 6275–6281, 2019.

[11] Sourabh Bhattacharya and Seth Hutchinson. Approximation schemes for two-player pursuit evasion games with visibility constraints. In *Robotics: Science and Systems*, 2008.

[12] Sourabh Bhattacharya and Seth Hutchinson. On the existence of nash equilibrium for a two player pursuit-evasion game with visibility constraints. In *Algorithmic Foundation of Robotics VIII*, pages 251–265. Springer, 2009.

[13] Sourabh Bhattacharya and Seth Hutchinson. A cell decomposition approach to visibility-based pursuit evasion among obstacles. *The International Journal of Robotics Research*, 30(14):1709–1727, 2011.

[14] Andreas Bircher, Mina Kamel, Kostas Alexis, Helen Oleynikova, and Roland Siegwart. Receding horizon" next-best-view" planner for 3d exploration. In *2016 IEEE international conference on robotics and automation (ICRA)*, pages 1462–1468. IEEE, 2016.

[15] Andreas Bircher, Mina Kamel, Kostas Alexis, Helen Oleynikova, and Roland Siegwart. Receding horizon path planning for 3d exploration and surface inspection. *Autonomous Robots*, 42(2):291–306, 2018.

[16] Iliana Bjorling-Sachs and Diane L. Souvaine. An efficient algorithm for guard placement in polygons with holes. *Discrete & Computational Geometry*, 13(1):77–109, 1995.

[17] Elliot Cartee, Lexiao Lai, Qianli Song, and Alexander Vladimirsky. Time-dependent surveillance-evasion games. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 7128–7133, 2019.

[18] Angel X Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, et al. Shapenet: An information-rich 3D model repository. *arXiv preprint arXiv:1512.03012*, 2015.

[19] Yat Tin Chow, Jérôme Darbon, Stanley Osher, and Wotao Yin. Algorithm for overcoming the curse of dimensionality for time-dependent non-convex hamilton–jacobi equations arising from optimal control and differential games problems. *Journal of Scientific Computing*, 73(2-3):617–643, 2017.

[20] Yat Tin Chow, Jérôme Darbon, Stanley Osher, and Wotao Yin. Algorithm for overcoming the curse of dimensionality for certain non-convex hamilton–jacobi equations, projections and differential games. *Annals of Mathematical Sciences and Applications*, 3(2):369–403, 2018.

[21] Yat Tin Chow, Jérôme Darbon, Stanley Osher, and Wotao Yin. Algorithm for overcoming the curse of dimensionality for state-dependent hamilton-jacobi equations. *Journal of Computational Physics*, 387:376–409, 2019.

[22] Václav Chvátal. A combinatorial theorem in plane geometry. *Journal of Combinatorial Theory*, B(18):39–41, 1975.

[23] Michael G Crandall and Pierre-Louis Lions. Viscosity solutions of hamilton-jacobi equations. *Transactions of the American mathematical society*, 277(1):1–42, 1983.

[24] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[25] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part i. *IEEE robotics & automation magazine*, 13(2):99–110, 2006.

[26] Lawrence C Evans and Panagiotis E Souganidis. Differential games and representation formulas for solutions of hamilton-jacobi-isaacs equations. *Indiana University mathematics journal*, 33(5):773–797, 1984.

[27] Yi Fang, Jin Xie, Guoxian Dai, Meng Wang, Fan Zhu, Tiantian Xu, and Edward Wong. 3D deep shape descriptor. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2319–2328, 2015.

[28] Fei Xia and others. Pointnet.pytorch git repository. `https://github.com/fxia22/pointnet.pytorch`.

[29] Subir K. Ghosh, Joel W. Burdick, Amitava Bhattacharya, and Sudeep Sarkar. Online algorithms with discrete visibility - exploring unknown polygonal environments. *IEEE Robotics Automation Magazine*, 15(2):67–76, June 2008.

[30] Marc Aurèle Gilles and Alexander Vladimirsky. Evasive path planning under surveillance uncertainty. *Dynamic Games and Applications*, pages 1–26, 2019.

[31] Héctor H González-Banos and Jean-Claude Latombe. Navigation strategies for exploring indoor environments. *The International Journal of*

Robotics Research, 21(10-11):829–848, 2002.

[32] Rostislav Goroshin, Quyen Huynh, and Hao-Min Zhou. Approximate solutions to several visibility optimization problems. *Communications in Mathematical Sciences*, 9(2):535–550, 2011.

[33] Leonidas J Guibas, Jean-Claude Latombe, Steven M LaValle, David Lin, and Rajeev Motwani. Visibility-based pursuit-evasion in a polygonal environment. In *Workshop on Algorithms and Data Structures*, pages 17–30. Springer, 1997.

[34] Markus Hadwiger, Christian Sigg, Henning Scharsach, Khatja Bühler, and Markus Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum*, 2005.

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[36] Lionel Heng, Alkis Gotovos, Andreas Krause, and Marc Pollefeys. Efficient visual exploration and coverage with a micro aerial vehicle in unknown environments. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1071–1078. IEEE, 2015.

[37] Yu-Chi Ho, Arthur Bryson, and Sheldon Baron. Differential games and optimal pursuit-evasion strategies. *IEEE Transactions on Automatic Control*, 10(4):385–389, 1965.

[38] Frank Hoffmann, Michael Kaufmann, and Klaus Kriegel. The art gallery theorem for polygons with holes. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 39–48. IEEE, 1991.

[39] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[40] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[41] Rufus Isaacs. *Differential games*. John Wiley and Sons, 1965.

[42] Peter Wilcox Jones, Andrei Osipov, and Vladimir Rokhlin. Randomized approximate nearest neighbors algorithm. *Proc. Natl. Acad. Sci.*, 2011.

[43] Sung Ha Kang, Seong Jun Kim, and Haomin Zhou. Optimal sensor positioning; a probability perspective study. *SIAM Journal on Scientific Computing*, 39(5):B759–B777, 2017.

[44] Wei Kang and Lucas C Wilcox. Mitigating the curse of dimensionality: sparse grid characteristics method for optimal feedback control and hjb equations. *Computational Optimization and Applications*, 68(2):289–315, 2017.

[45] Nikhil Karnad and Volkan Isler. Lion and man game in the presence of a circular obstacle. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5045–5050. IEEE, 2009.

[46] Ismail Khalid Kazmi, Lihua You, and Jian Jun Zhang. A survey of 2D and 3D shape descriptors. In *2013 10th International Conference Computer Graphics, Imaging and Visualization*, pages 1–10. IEEE, 2013.

[47] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[48] Daniel A Klain, Gian-Carlo Rota, et al. *Introduction to geometric probability*. Cambridge University Press, 1997.

[49] Roman Klokov and Victor Lempitsky. Escape from cells: Deep KD-networks for the recognition of 3D point cloud models. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 863–872, 2017.

[50] Andreas Krause and Daniel Golovin. Submodular function maximization. In *Tractability: Practical Approaches to Hard Problems*, pages 71–104. Cambridge University Press, 2014.

[51] Scott Krig. Interest point detector and feature descriptor survey. In *Computer vision metrics*, pages 187–246. Springer, 2016.

[52] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[53] Jens Kruger and Rüdiger Westermann. Acceleration techniques for gpu-based volume rendering. In *IEEE Visualization VIS 2003.*, 2003.

[54] Yanina Landa, David Galkowski, Yuan R Huang, Abhijeet Joshi, Christine Lee, Kevin K Leung, Gitendra Malla, Jennifer Treanor, Vlad Voroninski, Andrea L Bertozzi, Yen-Hsi Richard Tsai, et al. Robotic path planning and visibility with limited sensor data. In *American Control Conference, 2007. ACC'07*, pages 5425–5430. IEEE, 2007.

[55] Yanina Landa and Yen-Hsi Richard Tsai. Visibility of point clouds and exploratory path planning in unknown environments. *Communications in Mathematical Sciences*, 6(4):881–913, 2008.

[56] Yanina Landa, Yen-Hsi Richard Tsai, and Li-Tien Cheng. Visibility of point clouds and mapping of unknown environments. In *International Conference on Advanced Concepts for Intelligent Vision Systems*, pages 1014–1025. Springer, 2006.

[57] Steven LaValle. *Planning algorithms*. Cambridge University Press, 2006.

[58] Steven M LaValle, Hector H González-Banos, Craig Becker, and J-C Latombe. Motion strategies for maintaining visibility of a moving target.

In *Proceedings of International Conference on Robotics and Automation*, volume 1, pages 731–736. IEEE, 1997.

[59] Steven M LaValle and John E Hinrichsen. Visibility-based pursuit-evasion: The case of curved environments. *IEEE Transactions on Robotics and Automation*, 17(2):196–202, 2001.

[60] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[61] Der-Tsai Lee and Arthur Lin. Computational complexity of art gallery problems. *IEEE Transactions on Information Theory*, 32(2):276–282, 1986.

[62] Tai Lei and Liu Ming. A robot exploration strategy based on q-learning network. In *2016 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, pages 57–62. IEEE, 2016.

[63] Joseph Lewin and John Breakwell. The surveillance-evasion game of degree. *Journal of Optimization Theory and Applications*, 16(3-4):339–353, 1975.

[64] Li, Chen, and Hee Lee. SO-Net: self-organizing network for point cloud analysis. In *CVPR*, 2018.

[65] Yangyan Li, Rui Bu, Mingchao Sun, Wei Wu, Xinhan Di, and Baoquan Chen. Pointcnn: Convolution on x-transformed points. In *Advances in Neural Information Processing Systems*, pages 820–830, 2018.

[66] Louis Ly and Yen-Hsi Richard Tsai. Autonomous exploration, reconstruction, and surveillance of 3d environments aided by deep learning. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 5467–5473, © 2019 IEEE.

[67] Colin B Macdonald and Steven J Ruuth. The implicit closest point method for the numerical solution of partial differential equations on surfaces. *SIAM Journal on Scientific Computing*, 31(6):4330–4350, 2010.

[68] Emmanuel Maggiori, Yuliya Tarabalka, Guillaume Charpiat, and Pierre Alliez. Can semantic labeling methods generalize to any city? the inria aerial image labeling benchmark. In *IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*. IEEE, 2017.

[69] Antony W Merz. The homicidal chauffeur. *AIAA Journal*, 12(3):259–260, 1974.

[70] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[71] George L Nemhauser and Laurence A Wolsey. Best algorithms for approximating the maximum of a submodular set function. *Mathematics of operations research*, 3(3):177–188, 1978.

[72] Joseph O'Rourke. *Art gallery theorems and algorithms*, volume 57. Oxford University Press Oxford, 1987.

[73] Joseph O'Rourke and Kenneth Supowit. Some np-hard polygon decomposition problems. *IEEE Transactions on Information Theory*, 29(2):181–190, 1983.

[74] Stanley Osher and Ronald Fedkiw. *Level set methods and dynamic implicit surfaces*, volume 153. Springer Science & Business Media, 2006.

[75] Stanley Osher and James Albert Sethian. Fronts propagating with curvature-dependent speed: algorithms based on hamilton-jacobi formulations. *Journal of computational physics*, 79(1):12–49, 1988.

[76] Jon Peddie. *Ray Tracing: A Tool for All*. Springer, 2019.

[77] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 652–660, 2017.

[78] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In *Advances in neural information processing systems*, pages 5099–5108, 2017.

[79] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[80] Christopher D Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.

[81] Reihaneh Rostami, Fereshteh S Bashiri, Behrouz Rostami, and Zeyun Yu. A survey on data-driven 3D shape descriptors. *Comput. Graph. Forum*, 38(1):356–393, 2019.

[82] Steven J Ruuth and Barry Merriman. A simple embedding method for solving partial differential equations on surfaces. *Journal of Computational Physics*, 227(3):1943–1961, 2008.

[83] Shai Sachs, Steven M LaValle, and Stjepan Rajko. Visibility-based pursuit-evasion in an unknown planar environment. *The International Journal of Robotics Research*, 23(1):3–26, 2004.

[84] Nima Sedaghat, Mohammadreza Zolfaghari, Ehsan Amiri, and Thomas Brox. Orientation-boosted voxel nets for 3d object recognition. *arXiv preprint arXiv:1604.03351*, 2016.

[85] James Albert Sethian. *Level set methods and fast marching methods: evolving interfaces in computational geometry, fluid mechanics, computer vision, and materials science*, volume 3. Cambridge university press, 1999.

[86] Burr Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009.

[87] Jiří Sgall. Solution of david gale's lion and man problem. *Theoretical Computer Science*, 259(1-2):663–670, 2001.

[88] Yiru Shen, Chen Feng, Yaoqing Yang, and Dong Tian. Mining point cloud local structures by kernel correlation and graph pooling. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[89] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[90] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[91] Martin Simonovsky and Nikos Komodakis. Dynamic edge-conditioned filters in convolutional neural networks on graphs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3693–3702, 2017.

[92] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[93] Ayan Sinha, Jing Bai, and Karthik Ramani. Deep learning 3d shape surfaces using geometry images. In *European Conference on Computer Vision*. Springer, 2016.

[94] Nicholas M Stiffler and Jason M O'Kane. A complete algorithm for visibility-based pursuit-evasion with multiple pursuers. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1660–1667. IEEE, 2014.

[95] Hartmut Surmann, Andreas Nüchter, and Joachim Hertzberg. An autonomous mobile robot with a 3d laser range finder for 3d exploration and digitalization of indoor environments. *Robotics and Autonomous Systems*, 45(3-4):181–198, 2003.

[96] Ichiro Suzuki and Masafumi Yamashita. Searching for a mobile intruder in a polygonal region. *SIAM Journal on computing*, 21(5):863–888, 1992.

[97] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[98] Lei Tai and Ming Liu. Mobile robots exploration through cnn-based reinforcement learning. *Robotics and biomimetics*, 3(1):24, 2016.

[99] Lei Tai, Giuseppe Paolo, and Ming Liu. Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 31–36. IEEE, 2017.

[100] Ryo Takei, Weiyan Chen, Zachary Clawson, Slav Kirov, and Alexander Vladimirsky. Optimal control with budget constraints and resets. *SIAM Journal on Control and Optimization*, 53(2):712–744, 2015.

[101] Ryo Takei, Richard Tsai, Zhengyuan Zhou, and Yanina Landa. An efficient algorithm for a visibility-based surveillance-evasion game. *Communications in Mathematical Sciences*, 12(7):1303–1327, 2014.

[102] Benjamin Tovar, Luis Guilamo, and Steven LaValle. Gap navigation trees: Minimal representation for visibility-based tasks. In *Algorithmic Foundations of Robotics VI*, pages 425–440. Springer, 2004.

[103] Benjamín Tovar and Steven M LaValle. Visibility-based pursuitevasion with bounded speed. *The International Journal of Robotics Research*, 27(11-12):1350–1360, 2008.

[104] Benjamin Tovar, Rafael Murrieta-Cid, and Steven LaValle. Distance-optimal navigation in an unknown environment without sensing distances. *IEEE Transactions on Robotics*, 23(3):506–518, 2007.

[105] Yen-Hsi Richard Tsai. Rapid and accurate computation of the distance function using grids. *J. Comput. Phys.*, 2002.

[106] Yen-Hsi Richard Tsai, Li-Tien Cheng, Stanley Osher, Paul Burchard, and Guillermo Sapiro. Visibility and its dynamics in a pde based implicit framework. *Journal of Computational Physics*, 199(1):260–290, 2004.

[107] Yen-Hsi Richard Tsai, Li-Tien Cheng, Stanley Osher, and Hong-Kai Zhao. Fast sweeping algorithms for a class of hamilton–jacobi equations. *SIAM journal on numerical analysis*, 41(2):673–694, 2003.

[108] Yen-Hsi Richard Tsai and Stanley Osher. Total variation and level set methods in image science. *Acta Numerica*, 14:509–573, 2005.

[109] John N Tsitsiklis. Efficient algorithms for globally optimal trajectories. *IEEE Transactions on Automatic Control*, 40(9):1528–1538, 1995.

[110] Jorge Urrutia. Art gallery and illumination problems. In *Handbook of Computational Geometry*, pages 973–1027. Elsevier, 2000.

[111] Luca Valente, Yen-Hsi Richard Tsai, and Stefano Soatto. Information-seeking control under visibility-based uncertainty. *Journal of Mathematical Imaging and Vision*, 48(2):339–358, 2014.

[112] Peng-Shuai Wang, Yang Liu, Yu-Xiao Guo, Chun-Yu Sun, and Xin Tong. O-cnn: Octree-based convolutional neural networks for 3d shape analysis. *ACM Transactions on Graphics (TOG)*, 36(4):72, 2017.

[113] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. Dynamic graph cnn for learning on point clouds. *ACM Transactions on Graphics (TOG)*, 38(5):146, 2019.

[114] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets: A deep representation for volumetric shapes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1912–1920, 2015.

[115] Jin Xie, Guoxian Dai, Fan Zhu, Edward K Wong, and Yi Fang. Deepshape: Deep-learned shape descriptor for 3d shape retrieval. *IEEE transactions on pattern analysis and machine intelligence*, 39(7):1335–1345, 2016.

[116] Brian Yamauchi. A frontier-based approach for autonomous exploration. In *Computational Intelligence in Robotics and Automation, 1997. CIRA'97., Proceedings., 1997 IEEE International Symposium on*, pages 146–151. IEEE, 1997.

[117] Fumin Zhang, Alan O'Connor, Derek Luebke, and PS Krishnaprasad. Experimental study of curvature-based control laws for obstacle avoidance. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 4, pages 3849–3854. IEEE, 2004.

[118] Mengzhe Zhang and Sourabh Bhattacharya. Multi-agent visibility-based target tracking game. In *Distributed Autonomous Robotic Systems*, pages 271–284. Springer, 2016.

[119] Ding-Xuan Zhou. Universality of deep convolutional neural networks. *Applied and computational harmonic analysis*, 48(2):787–794, 2020.

[120] Yin Zhou and Oncel Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4490–4499, 2018.

[121] Rui Zou and Sourabh Bhattacharya. Visibility-based finite-horizon target tracking game. *IEEE Robotics and Automation Letters*, 1(1):399–406, 2016.

[122] Rui Zou and Sourabh Bhattacharya. On optimal pursuit trajectories for visibility-based target-tracking game. *IEEE Transactions on Robotics*, 35(2):449–465, 2018.

[123] Rui Zou, Hamid Emadi, and Sourabh Bhattacharya. On the optimal policies for visibility-based target tracking. *arXiv preprint arXiv:1611.04613*, 2016.